REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
05-09-2014		

4. TITLE AND SUBTITLE

System Identification and Control of a Joint-Actuated Buoy

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Fugleberg, Eric Nels

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 426 (2014)

12. DISTRIBUTION / AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Consider the example of a small, free-floating buoy using a directional antenna to communicate with a satellite. The position of the buoy must remain stable and directed towards the satellite for effective transmission. With a joint-actuated buoy, it is mechanically possible to stabilize the free-floating buoy and the antenna. The goal of this Trident project is to perform system identification of the joint-actuated buoy; prepare an animation for the response of the buoy to different inputs; and create a single loop control law for the two dimensional control of the buoy to keep it vertical at all times. To complement theoretical models derived from Newtonian physics, an experimentally derived model allows non-linear effects such as drag and added mass effects to be inherently accounted for within the experimental data. Using step response testing for the top and bottom servos was the primary method used to formulate the experimental model. An animation was created to help visually comprehend the dynamics of the buoy. Next, a single loop control law was developed to control the position of the buoy in two dimensions to keep the buoy vertical at all times; keeping the buoy stable while at a specified angle off of vertical was not attempted in this project due to the complexities of three dimension control with only two control surfaces.

15. SUBJECT TERMS
system identification, experimental modeling, free-floating buoy, control systems, joint-actuated

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		52	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

U.S.N.A. --- Trident Scholar project report; no. 426 (2014)

System Identification and Control of a Joint-Actuated Buoy

by

Midshipman 1/C Eric N. Fugleberg
United States Naval Academy
Annapolis, Maryland

_____
(signature)

Certification of Adviser(s) Approval

CAPT Owen G. Thorp, USN
Weapons and Systems Engineering Department

_____
(signature)

_____
(date)

Professor George E. Piper
Weapons and Systems Engineering

_____
(signature)

_____
(date)

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

_____
(signature)

_____
(date)

USNA-1531-2

Abstract

Consider the example of a small, free-floating buoy using a directional antenna to communicate with a satellite. The position of the buoy must remain stable and directed towards the satellite for effective transmission. With a joint-actuated buoy, it is mechanically possible to stabilize the free-floating buoy and the antenna. The goal of this Trident project is to perform system identification of the joint-actuated buoy; prepare an animation for the response of the buoy to different inputs; and create a single loop control law for the two dimensional control of the buoy to keep it vertical at all times.

To complement theoretical models derived from Newtonian physics, an experimentally derived model allows non-linear effects such as drag and added mass effects to be inherently accounted for within the experimental data. Using step response testing for the top and bottom servos was the primary method used to formulate the experimental model. An animation was created to help visually comprehend the dynamics of the buoy. Next, a single loop control law was developed to control the position of the buoy in two dimensions to keep the buoy vertical at all times; keeping the buoy stable while at a specified angle off of vertical was not attempted in this project due to the complexities of three dimension control with only two control surfaces.

The system identification model, simulation, and single loop control law for the buoy could be utilized by the Navy to monitor meteorological measurements of the air column above the ocean surface, directional communications with a satellite, and support many other diverse operations.

Keywords: system identification, experimental modeling, free-floating buoy, control systems, joint-actuated

Acknowledgements

I would like to thank the following people for assisting me, for without their help I could not have successfully completed this project:

My advisors, CAPT Owen G. Thorp, USN and Professor George Piper, for their continued guidance, support, and motivation.

Dr. Roger Cortesi and his team at Naval Research Lab for the project inspiration and mentorship.

Professor Jenelle Piepmeier for her crash course on computer vision and MATLAB coding to manipulate images for spacial recognition.

Joe Bradshaw and Norm Tyson in TSD for their patience and assistance after knocking on their doors over a hundred times during the year. Without these two gentlemen, I could not have completed the more intricate design and hardware aspects of this project.

Roy Goddard and the team in the Machine Shop for their efforts in assisting with fabrication of buoy parts.

Table of Contents

List of Appendices

List of Figures

Chapter 1: Introduction

1.1 Problem Statement

The goal of this Trident project was to experimentally derive a mathematical model of a joint-actuated, free-floating buoy; prepare a high fidelity animation for the response of the buoy; and a develop single loop control law to keep the buoy vertical despite wave disturbances. Using an existing prototype, step response testing was the primary method used to create the experimental model.

1.2 Motivation

Consider the example of a small, free-floating buoy using a directional antenna with a beam width of ten degrees to communicate with a satellite using a remote sending application, as shown in Figure 1. To communicate effectively, the position of the buoy must remain stable and directed towards the satellite within the performance objective of the half power point of the satellite antenna, five degrees. With a joint-actuated buoy, it is mechanically possible to stabilize the free-floating buoy and the antenna. Payloads such as a camera or some other sensor with a narrow field of view could be used as well. Such a buoy presents many assets for the United States Navy and the mission of reconnaissance and intelligence. Attaching a camera or some

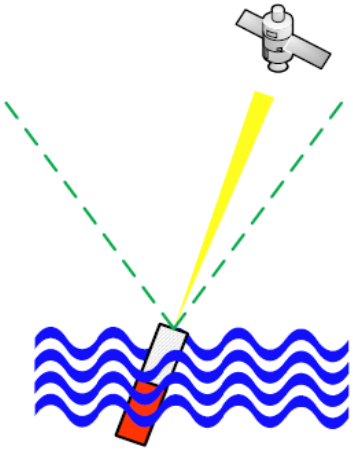


Figure 1: Joint-Actuated Buoy Directed Towards a Satellite

other narrow field of view device could allow meteorological measurements of the air column above the ocean surface or provide directional communications with a satellite. Additionally, the design of the buoy complies with NATO size A sonobuoy dimensions, shown in Figure 2, making it compatible with any sonobuoy platform used by the Navy. A joint-actuated buoy is comprised of two distinct sections: the payload and the buoy housing. The payload contains the directional antenna and is the upper portion of the buoy, while the housing is the lower portion, shown in Figure 2.

	Size A Sonobuoy (in)	Join-Actuated Buoy (in)
Length	36	32.36
Diameter	4.875	3.13

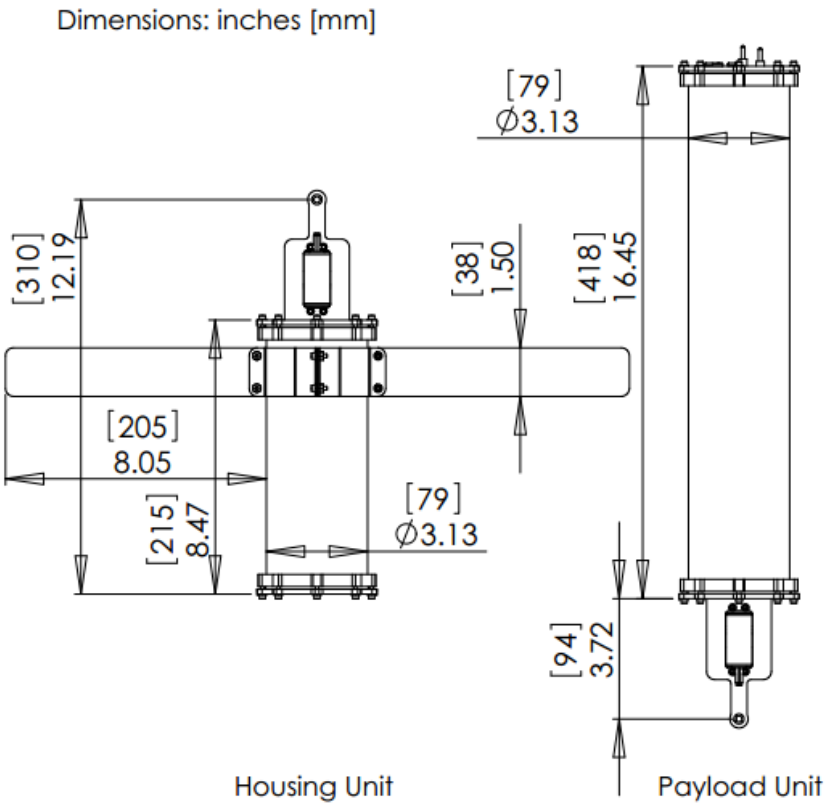


Figure 2: Size A Sonobuoy and Joint-Actuated Buoy Dimensions

1.3 Related Work

There are numerous publications describing directing antennas from floating or space objects. One method, as proposed by Timothy et al. [1], utilizes a pair or servomotors to direct the antenna towards its desired target. Two critical assumptions by Timothy are: the mass of the antenna is small compared to the mass of the vessel and the movement of the antenna caused by the servomotors does not impart any motion on the vessel. Frye et al. [2] have done work to point a directional antenna from a floating buoy. Their research evaluated the radio frequency design for the antenna, and it is also assumed that the movement of the antenna would not influence the motion of the vessel. Briguglio [3] proposed a multibody buoy to position an antenna with his Antenna Stabilizing Buoy invention. However, since Briguglio's invention is comprised of several buoys connected by a four bar linkage, it is not practical for the proposed scenario. M. Romano and B. N. Agrawal [4] worked on a multibody spacecraft in which the two bodies were connected by a one degree of freedom joint. The intention was to use the satellite to redirect a ground-based laser from one ground station to another. While similar to the design of the buoy,

the satellite system differs from the buoy system in three major ways: the spacecraft must have both bodies directed towards their respective targets, the spacecraft's center of mass is at the joint, and the joint is a single degree of freedom. For this project, only the payload must be pointed at the intended target, the center of mass is constantly changing to augment the position of the buoy, and the joint of the buoy has two degrees of freedom.

Dr. Roger Cortesi of Naval Research Laboratory (NRL) in Washington, D.C. has proposed a theoretical model for the joint-actuated buoy described in this scenario [5]. The model applies Newtonian physics, incorporates an eight degree of freedom system, and attempts to describe the motion of the buoy through mathematical derivations. The non-linear effects of drag and added mass effects are the most uncertain components of the model. An experimentally derived mathematical model allows a model to be constructed through the testing of buoy responses given different inputs. All the uncertainties of the theoretical model, primarily fluid dynamics around the buoy, are inherently accounted for in the experimental data.

Until recently, there has not been a prototype on which to perform experimental tests. Over the past year, Dr. Cortesi and his team of researchers have successfully developed but not tested a prototype of the proposed joint-actuated buoy, shown in Figure 3. It is with this buoy prototype that the experimental model was created. The goal of this project was to determine the experimental model of the buoy and create control laws to keep the buoy vertical at all time.

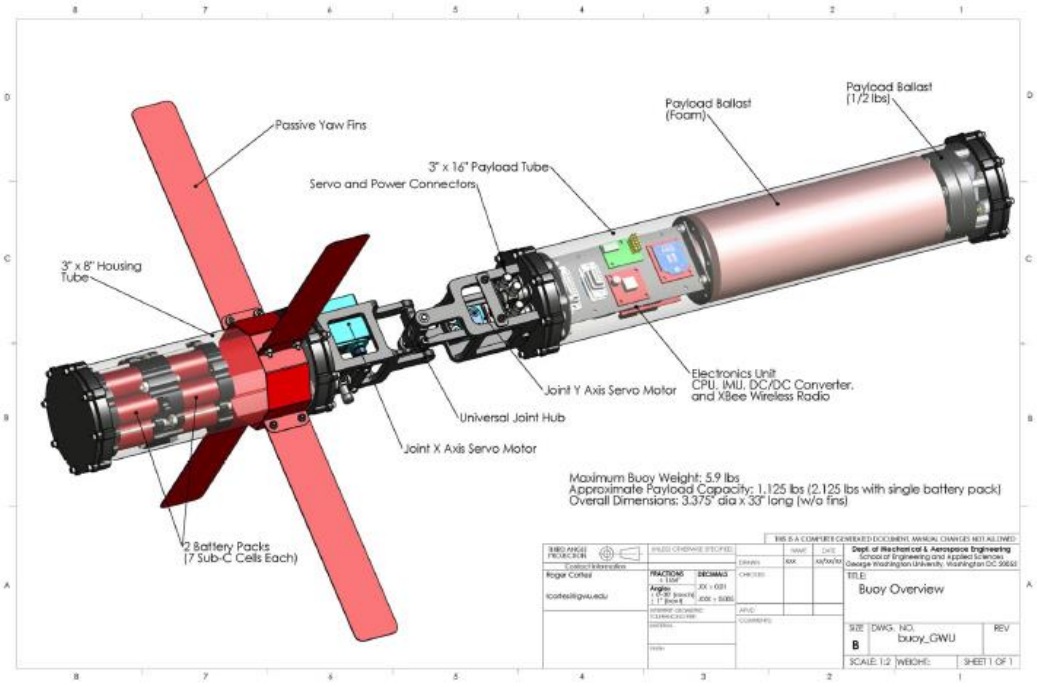


Figure 3: Joint-Actuated Buoy Prototype

Chapter 2: The Buoy

Dr. Cortesi and his team at NRL already fabricated many of the external components of the buoy joint before they lent the buoy for this project; this included the lengths of acrylic cylindrical tubing, aluminum end caps, and aluminum universal. The buoy was not watertight, and the internal components to control the buoy were not functional.

2.1 Waterproofing

The buoy design utilized an aluminum ring glued to the end of the acrylic tubing with the aluminum cap screwing onto the aluminum ring; inside the aluminum end cap were two O-rings to help create a watertight seal. Initial testing revealed that this design did not prove to be watertight with the water leaking between the aluminum ring and the acrylic tubing. Adding an acrylic flange below the aluminum ring was determined as the best solution because the bond between acrylic surfaces was watertight. The aluminum ring was sandwiched between the aluminum end cap and the acrylic flange, as show in Figure 4, to create a watertight seal.

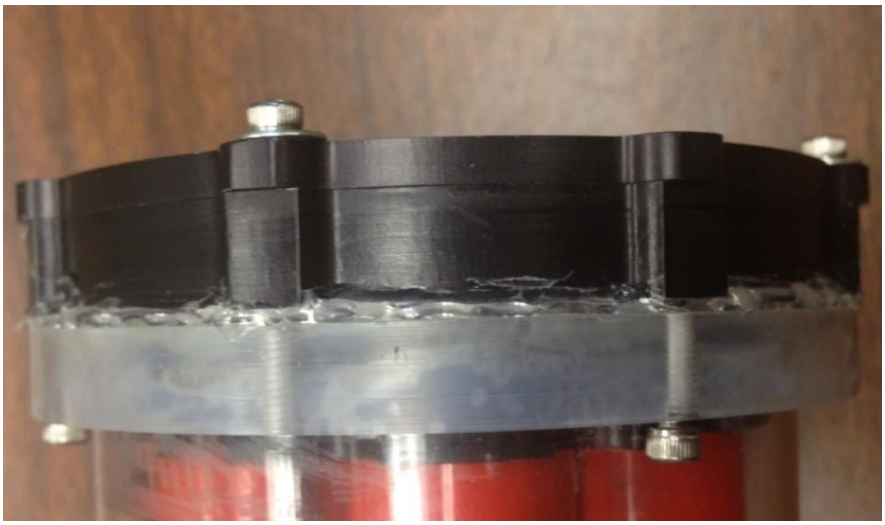


Figure 4: End Cap/Ring Assembly Attached to Acrylic Flange

To assemble the end cap, ring, and flange assembly properly required a specific method to ensure the watertight seal. First, the aluminum end cap and ring screwed together using four 3/8” #4 screws placed equidistant from one another to form an end cap/ring assembly. The gap between the end cap and the ring where the acrylic tubing fits, shown in Figure 5, was filled with acrylic glue. Once filled with glue, the end cap/ring assembly was placed onto the end of the buoy until it reached the acrylic flange. Four 3/8” #4 screws were then placed offset from the first four screws to connect the end cap/ring assembly to the acrylic flange. The glue cured after several hours and created the necessary watertight seal. This process was used on all four ends of the buoy – both ends of the payload and housing. Once the buoy was watertight, the next step was to organize the internal components for testing.

Figure 5: Aluminum End Cap and Ring

2.2 Internal Hardware

The internal hardware for the joint-actuated buoy included a power supply, processor, inertial measurement unit (IMU), wireless transmitter, and actuators.

2.2.1 Power Supply

The buoy used two battery packs, each containing seven sub-C cell batteries, for fourteen Nickel-Cadmium batteries total. These motivational fourteen cells provided an average of 8.5 volts to the buoy powering the processor, IMU, wireless transmitter, and the actuators.

2.2.2 Processor

An Arduino Mega 2560 was the processor of choice. It has 54 digital input/output pins, of which 15 can be used for pulse width modulation (PWM), 16 analog inputs, 4 hardware serial ports, and a USB connection. The four serial ports on the Arduino Mega 2560 were the deciding factor in choosing this processor; no other Arduino has as many serial ports to interface with other devices. This specific project required at least two serial ports for communication with the IMU and the wireless communication transmitter. The Arduino operated on 5 volts regulated from the batteries.

2.2.3 Inertial Measurement Unit (IMU)

The inertial measurement unit used was the VectorNav VN-100 Rugged. This miniature, high-performance IMU combines 3-axis accelerometers, 3-axis gyros, 3-axis magnetic sensors, and a 32-bit processor into a durable and lightweight three-dimensional and 360 degree position measurement sensor. Dimensions of 1.30 x 1.41 x 0.35 in. and a mass of 13 grams make the VN-

100 Rugged the smallest commercially available industrial grade IMU. Furthermore, the VN-100 uses a quaternion based Kalman filter, ensuring reliable operation during gimbal-lock – when two axes are driven into parallel. Update rates as quick as 200Hz ensures the VN-100 reported orientation with minimal latency, a crucial requirement for control systems. The IMU operated on 5 volts supplied by the Arduino.

2.2.4 Wireless Transmitter

The XBee RF modules provided wireless end-point connectivity to devices. The transmitting XBee was located within the buoy and the receiving XBee mounted on a USB shield to attach to the USB port on the computer used for data collection. With ranges up to 100 foot and interface data rates up to 115.2 Kbps, the XBee was perfect for testing and system identification to record and analyze the IMU data. The wireless transmitter operated on 3.3 volts provided by the Arduino.

2.2.5 Actuators

The joint was actuated using two Hitec HS-5646WP metal gear digital servos. These servos are compact in size, measuring 1.65 x 0.83 x 1.57 in., and waterproof. They are able to move 60 degrees in 0.20 seconds with 157 oz.-in of torque at six volts. This was important for the buoy application because the ability to quickly and efficiently change the position of the buoy housing is only second to the processor's quickness in sending commands to the servos. The actuators received their power directly from the batteries.

2.3 Basic Communication Flow

The buoy used the following communication protocol through the internal hardware to transmit orientation data from the IMU to a computer for data processing:

1. The Arduino sends a command string to the IMU requesting orientation data
2. The IMU sends the requested data to the Arduino
3. The Arduino reads the IMU string of data and parses the data for the two dimensional orientation of the buoy and the time. Additionally, the Arduino takes the IMU data and determines the proper command to send to the top and bottom servos.
4. The two orientation measurements and the time are forwarded to the XBee on board the buoy for transmission to the other XBee attached to a computer
5. Data is read on the computer using a serial monitoring program and stored in a .txt file for data processing

Appendix A lists the created Arduino code used to complete the objective above. Once the proper protocol was established for interfacing all the internal components, the components had to be assembled into a concise package to fit within the buoy payload.

2.4 Printed Circuit Board Construction

A princted circuit board (PCB) allowed all the internal components to neatly interact within one another. Creating the schematic for the printed circuit board required the use of the ExpressPCB software. Figure 6 shows the final circuit board ExpressPCB drawing; the green markings represent the drawings to be printed while the yellow marks are for aesthetic and orientation purposes. Working from top to bottom on Figure 6, the top serial port connection went to the top of the buoy payload. This allowws for the buoy to be opperated by an external power source and for the Arduino to be reprogrammed. Below the serial port, the MAX232 chip was a 5 volt regulator. As mentioned earlier, the batteries provides 8.5 volts to the buoy, but the Arduino operated on 5 volts. In the middle of the board was another serial port used to communicate with the IMU. Below the second serial port was a circuit breaker to help protect the IMU from any voltage surges. Lastly, at the base of the board was a third serial port which was used to receive power from the batteries and send commands to the servos. The small holes on the left, bottom, and right are the pin locations so the printed circuit board may be used as a shield and fit on top of the Arduino.

The green drawing was printed onto blue transfer paper and stuck to the copper side of a blank prefabricated circuit board. The copper and blue paper were then heated in a press-n-peel press for just over one minute to allow the printer toner to bond with the copper. The blue paper was removed, leaving only the printed circuit board design on the copper. The copper was then submerged in a ferric chloride etching solution. The solution disolved all exposed copper from the originally blank circuit board, leaving only copper that was covered by the toner from the circuit board design. Once all the excess copper dissolved, the toner was scrubbed off to reveal the circuit board. Holes were then drilled in the necessary spots for the serial ports and other components to be soltered into place. Figure 7 shows the completed circuit board ready to be mounted on top of the Arduino Mega 2560 processor. Figure 8 shows the PCB mounted onto the Arduino 2560 with the VectorNav IMU, XBee module, and serial connnections completed for installation within the buoy. With the internal hardware no installed, the next step was to begin data collection and start the step response testing.

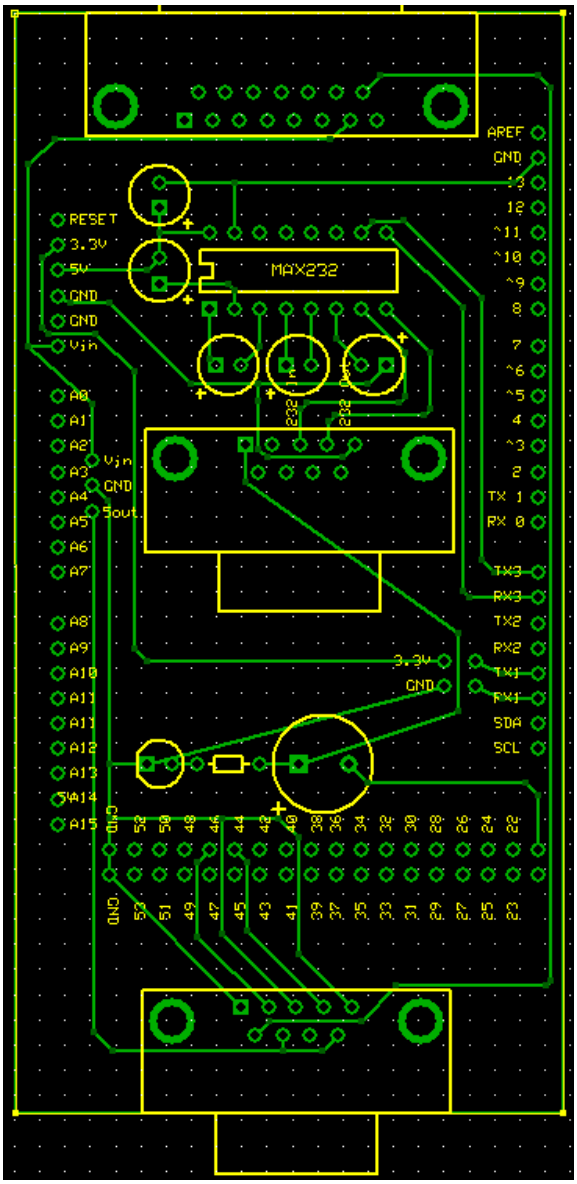


Figure 6: Circuit Board Schematic

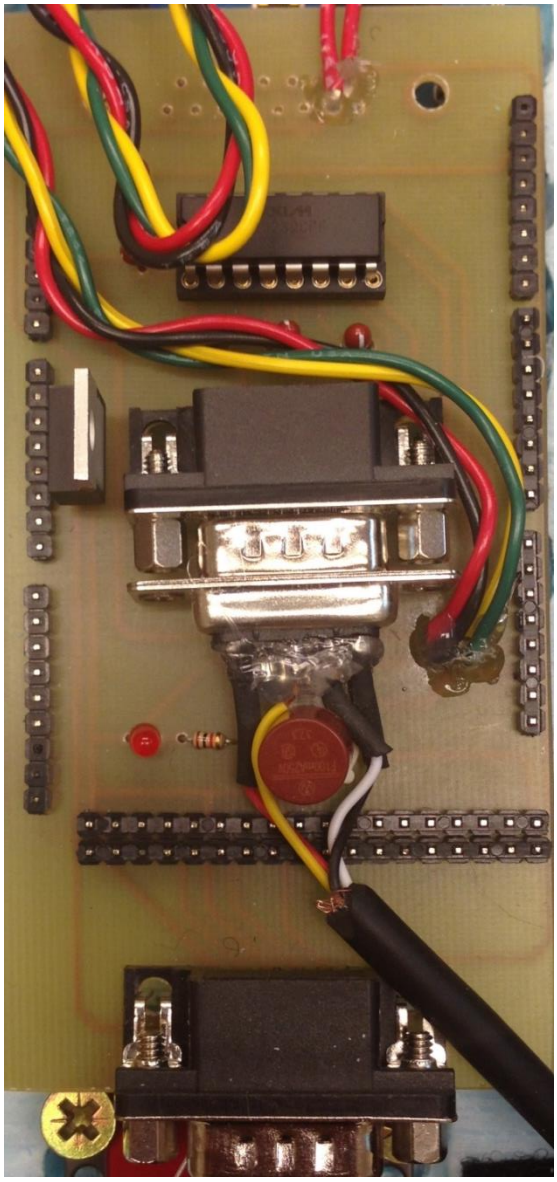


Figure 7: Completed Circuit Board

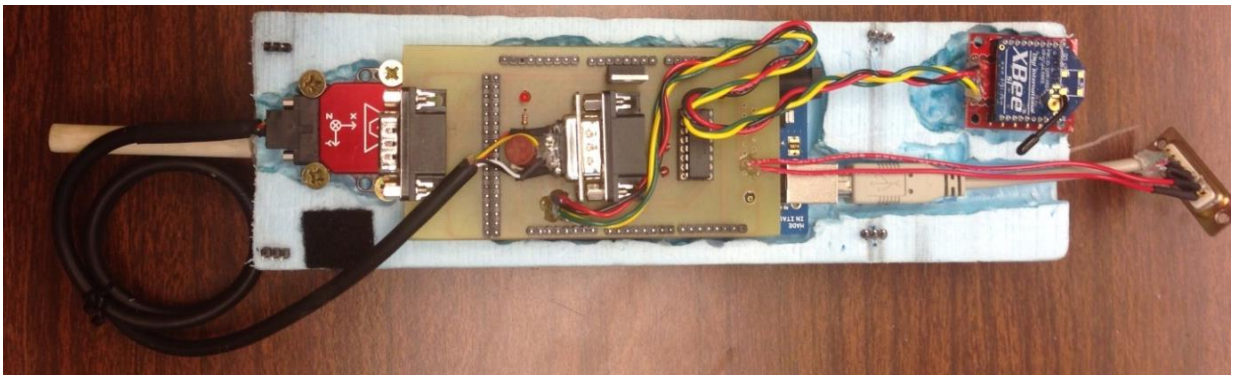


Figure 8: Final Assembly of Buoy Internal Components

Chapter 3: Data Collection

3.1 IMU Data

Data received from the IMU was recorded using an RS232 Port Logger program from Eltima software and stored as a .txt file in a specified directory. The IMU data was formatted in the Arduino program to send the pitch, roll, and time values to the computer. From this .txt file, MATLAB code plotted the two dimensional position of the buoy.

To monitor the position of the buoy in two-dimensional space, computer vision was intended as a secondary method. The two methods for orientation would have acted as redundancy to ensure the position data collected from the buoy was as accurate as possible. Electrical supply difficulties with the VectorNav IMU required making computer vision the primary method of determining the buoy orientation, as the IMU was deemed undependable for testing.

3.2 Computer Vision for Orientation

The other method for collecting buoy orientation data was tracking of colored markers placed on the outside of the acrylic tubing. The overall process required a camera, the camera to record the movement of the buoy, and then MATLAB code to analyze the video frames. The first step was to determine what camera and lens size to use. The Naval Academy Weapons and Systems Engineering Department had several blue Imaging Source cameras readily available for use. The calculated pinhole model focal length for a three-foot buoy observed from nine feet away with a 4.8 mm sensor was 14.4 mm. Based on readily available lenses within the Department a 12 mm lens was chosen.

Based on recommendations, children's craft foam was used for the markers. Their matte finish kept variations due to brightness at a minimum, and the colors of orange and green were chosen because there were no orange or green objects near the intended testing site that could be accidentally interpreted as the buoy. These colors were placed alternating and equidistant to each other running along the length of the payload, as shown in Figure 9.

Figure 9: Green and Orange Colored Stripes of Craft Foam on the Buoy Payload

3.3 Step Testing

With the ability to track the orientation of the buoy, step response testing began to determine the experimental model of the buoy. Testing was conducted in the Naval Academy 120 foot tow tank with the camera placed perpendicular to the tow tank to record essentially one dimension of orientation at a time. Tests were conducted using the two servos, top and bottom servo, which comprised the joint. Step tests were conducted by sending a command to first "center" the top and bottom servos to make the buoy vertical in two dimensions. Next, a step input was sent to the bottom servo to change the "center" position by +5 degrees. The proceeding buoy oscillations were recorded and stored as an AVI file for post testing data processing. The power was cycled, and the Arduino was reprogrammed to give the servo a +10 degree change the second time. The proceeding buoy oscillations were recorded and stored as an AVI file once again. This process was repeated at intervals of 5 degrees up to +35 degrees, and then with opposite commands from -5 to -35 for a total of 14 different step input tests for the bottom servo. The buoy was rotated 90 degrees to complete testing for the other servo because the camera could only record one dimension of orientation at a time. All 14 tests were repeated again with the top servo to produce 28 total step response tests. Figure 10 depicts the testing matrix and a graphic to help visualize a step input.

Top and Bottom Servo Step Inputs (degrees)													
-35	-30	-25	-20	-15	-10	-5	+5	+10	+15	+20	+25	+30	+35

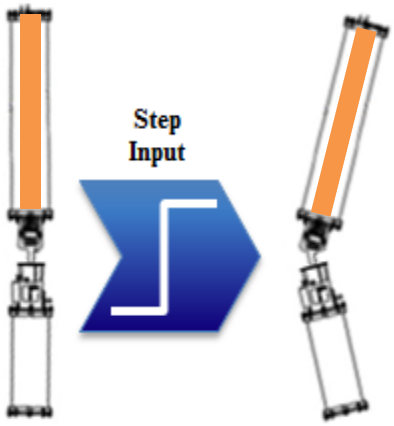


Figure 10: Buoy Step Response Testing Matrix and Graphic

Chapter 4: Data Processing

4.1 Image Processing

MATLAB code in Appendix D was used to analyze the videos on a frame by frame basis. The code read each video one frame at a time, created a red-blue-green (RGB) .jpg image of the frame, and saved the frame in a new folder. Once this was completed for all the frames of the video, the next step was to analyze each frame individually:

A. The code read each .jpg image of the buoy and converted the image from RGB to YCbCr color model to help filter out any disparities due to differing brightness.
    I. RGB is the red-green-blue spectrum that is most common in imaging; it is very susceptible to differences to brightness. YCbCr is a different color scheme that is based on the brightness. Y is the brightness (luma), Cb is the blue minus luma (B-Y), and Cr is red minus luma (R-Y). YCbCr is not as easily affected by changes in brightness and was the choice for computer vision analysis.

B. The YCbCr image was filtered to a black and white image where only pixels that fell within the desired orange or green YCbCr color parameters had their values changed to white; all other pixels values were changed to black.

C. The black and white image underwent another filter that deleted any groupings of pixels less than a desired size of 2400 pixels and created a second black and white image. The only displayed object was a white rectangle representing the orange or green strip of craft foam.

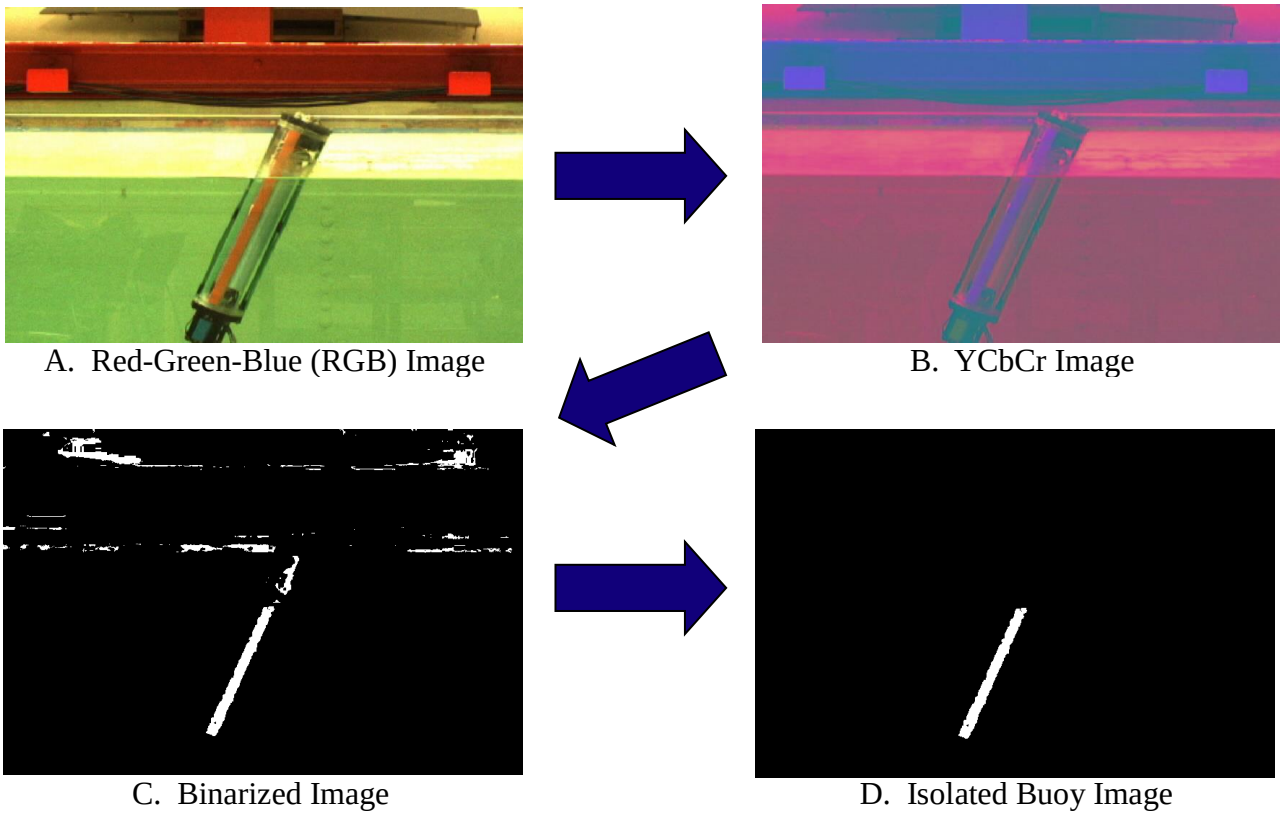
D. One last MATLAB function was used to determine the orientation of the white block with respect to the horizontal axis. The value was stored in a matrix of orientation values.

This four-step process, depicted in Figure 11, repeated for each .jpg image from all 28 videos. Appendix B depicts the created MATLAB code to complete the previously mentioned algorithm.

4.2 System Identification

Now that the buoy orientation had been calculated for each frame, the matrix of orientations were plotted to produce a plot of the step response in degrees versus time. Additionally, the MATLAB function *ssest* was used to derive the desired state space model. *Ssest* is a function that estimates a state-space model of a desired order using time domain data including input data, output data, and the sample time at which the data was recorded.

Using visual inspection for different orders of state space models, a fifth order state space model was the simplest model that could still encompass the buoy dynamics. The input data was the



A. Red-Green-Blue (RGB) Image

B. YCbCr Image

C. Binarized Image

D. Isolated Buoy Image

Figure 11: Image Processing for Each Frame of Video

servo command ranging from -35 to +35, the output data was the orientation of the buoy as calculated from the computer vision image processing, and the sample time was 30 frames per second, or 0.033 seconds. Appendix C depicts the code to create each state space model.

4.3 Modal Form and Eigenvalues

After producing the 28 different state space models, the next step was to determine the best model that described the response of the buoy. As the 28 fifth order state space models did not have any pattern of resemblance, there needed to be a way to compare all the individual experimental models to create one model to describe the buoy system. Converting the matrices to their modal canonical form allowed this ability.

To convert a matrix into modal form requires calculating and placing the eigenvalues on the diagonal. Solving Equation 1 for λ produces the eigenvalues for any *nxn* matrix *A*; there will be *n* eigenvalues for the *nxn* matrix.

$$det(A - \lambda I) = 0 \quad \textbf{(1)}$$

The next step is to place the eigenvalues diagonally in the new modal matrix M. For a system with the four eigenvalues, λ_1, $-\sigma \pm j\omega$, and λ_2, Equation 2 depicts the arrangement of the eigenvalues in the modal form.

$$M = \begin{bmatrix} -\lambda_1 & 0 & 0 & 0 \\ 0 & -\sigma & \omega & 0 \\ 0 & -\omega & -\sigma & 0 \\ 0 & 0 & 0 & -\lambda_2 \end{bmatrix} \quad \textbf{(2)}$$

The fifth order buoy, as such, had five eigenvalues: one real and two pairs of imaginary eigenvalues. Therefore, the common arrangement of each modal matrix A_M was:

$$A_M = \begin{bmatrix} -\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & -\sigma_1 & \omega_1 & 0 & 0 \\ 0 & -\omega_1 & -\sigma_1 & 0 & 0 \\ 0 & 0 & 0 & -\sigma_2 & \omega_2 \\ 0 & 0 & 0 & -\omega_2 & -\sigma_2 \end{bmatrix}$$

To complete the state space model, the B and C matrices must be accounted for as well [6].

$$B_M = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \qquad C_M = [c_1 \quad c_2 \quad c_3 \quad c_4 \quad c_5]$$

Once all the derived state space models had been placed in their modal form, an excel spreadsheet in Appendix E calculated the average of all the eigenvalues. Figure 12 shows the six models that were calculated: averages of bottom servo tests between -20 and +20 degrees, all bottom servo tests, top servo tests between -20 and +20 degrees, all top servo tests, all tests between -20 and +20 degrees, and all servo tests.

Bottom Servo Averages (-20<x<20)

$$A = \begin{bmatrix} -893.858 & 0 & 0 & 0 & 0 \\ 0 & -8.00003 & 53.93236 & 0 & 0 \\ 0 & -53.9324 & -8.00003 & 0 & 0 \\ 0 & 0 & 0 & -0.16218 & 1.191387 \\ 0 & 0 & 0 & -1.19139 & -0.16218 \end{bmatrix}$$

$$B = \begin{bmatrix} 2.711915 \\ -1.00717 \\ 0.579117 \\ -0.29772 \\ 0.11578 \end{bmatrix}$$

$$C = \begin{bmatrix} 4.032176 & 0.311822 & 0.006222 & 8.377624 & 2.579287 \end{bmatrix}$$

Model 1

Bottom Servo Averages (-35<x<35)

$$A = \begin{bmatrix} -609.862 & 0 & 0 & 0 & 0 \\ 0 & -12.2129 & 64.1822 & 0 & 0 \\ 0 & -64.1822 & -12.2129 & 0 & 0 \\ 0 & 0 & 0 & -0.1878 & 1.160011 \\ 0 & 0 & 0 & -1.16001 & -0.1878 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.107122 \\ -1.94778 \\ -0.1773 \\ 0.473614 \\ 0.094244 \end{bmatrix}$$

$$C = \begin{bmatrix} 2.458298 & 1.056931 & -0.29381 & 4.224527 & 0.607754 \end{bmatrix}$$

Model 2

Top Servo Averages (-20<x<20)

$$A = \begin{bmatrix} -71.8978 & 0 & 0 & 0 & 0 \\ 0 & -4.82904 & 81.78592 & 0 & 0 \\ 0 & -81.7859 & -4.82904 & 0 & 0 \\ 0 & 0 & 0 & -0.14611 & 1.133906 \\ 0 & 0 & 0 & -1.13391 & -0.14611 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.881358 \\ 0.16733 \\ -0.16848 \\ 0.082874 \\ 0.208461 \end{bmatrix}$$

$$C = \begin{bmatrix} -1.14287 & -0.23121 & 0.129898 & 1.611007 & -1.70041 \end{bmatrix}$$

Model 3

Top Servo Averages (-35<x<35)

$$A = \begin{bmatrix} -121.551 & 0 & 0 & 0 & 0 \\ 0 & -8.91663 & 59.59124 & 0 & 0 \\ 0 & -59.5912 & -8.91663 & 0 & 0 \\ 0 & 0 & 0 & -0.18976 & 1.113754 \\ 0 & 0 & 0 & -1.11375 & -0.18976 \end{bmatrix}$$

$$B = \begin{bmatrix} 3.452307 \\ 0.077058 \\ 0.957669 \\ -0.31222 \\ -0.41615 \end{bmatrix}$$

$$C = \begin{bmatrix} -0.68578 & 0.821313 & 0.41537 & -0.15765 & 1.571224 \end{bmatrix}$$

Model 4

Averages of ALL Tests (-20<x<20)

$$A = \begin{bmatrix} -482.878 & 0 & 0 & 0 & 0 \\ 0 & -6.41454 & 67.85914 & 0 & 0 \\ 0 & -67.8591 & -6.41454 & 0 & 0 \\ 0 & 0 & 0 & -0.15415 & 1.162646 \\ 0 & 0 & 0 & -1.16265 & -0.15415 \end{bmatrix}$$

$$B = \begin{bmatrix} 1.796636 \\ -0.41992 \\ 0.20532 \\ -0.10743 \\ 0.16212 \end{bmatrix}$$

$$C = \begin{bmatrix} 2.587525 & 0.271517 & -0.06184 & 3.383309 & 2.139851 \end{bmatrix}$$

Model 5

Averages of ALL Tests (-35<x<35)

$$A = \begin{bmatrix} -365.707 & 0 & 0 & 0 & 0 \\ 0 & -10.5648 & 61.88672 & 0 & 0 \\ 0 & -61.8867 & -10.5648 & 0 & 0 \\ 0 & 0 & 0 & -0.18878 & 1.136882 \\ 0 & 0 & 0 & -1.13688 & -0.18878 \end{bmatrix}$$

$$B = \begin{bmatrix} 1.779714 \\ -0.93536 \\ 0.390182 \\ 0.080695 \\ -0.16095 \end{bmatrix}$$

$$C = \begin{bmatrix} 0.88626 & 0.939122 & 0.06078 & 2.033436 & 1.089489 \end{bmatrix}$$

Model 6

Figure 12: Each of the Six Different Models

Chapter 5: Modeling and Animation

5.1 Modeling

MATLAB code in Appendix F was used to simulate each state space model by giving it a step input of +20 degrees, shown in Figure 13. The two main factors for determining the performance of a model was the amount of time until it made movement in the intended direction and how large was the peak overshoot. Models 1, 3, and 4 went opposite of the intended direction for too long, and Models 2 and 6 had a peak overshoot of nearly 100% when compared to their steady state value. That left Model 5, created averaging all tests conducted between -20 and +20 servo commands, as the best performing model out of the group. It should be noted that while each figure had different response amplitudes, all possessed similar dampening and oscillation frequency characteristics.

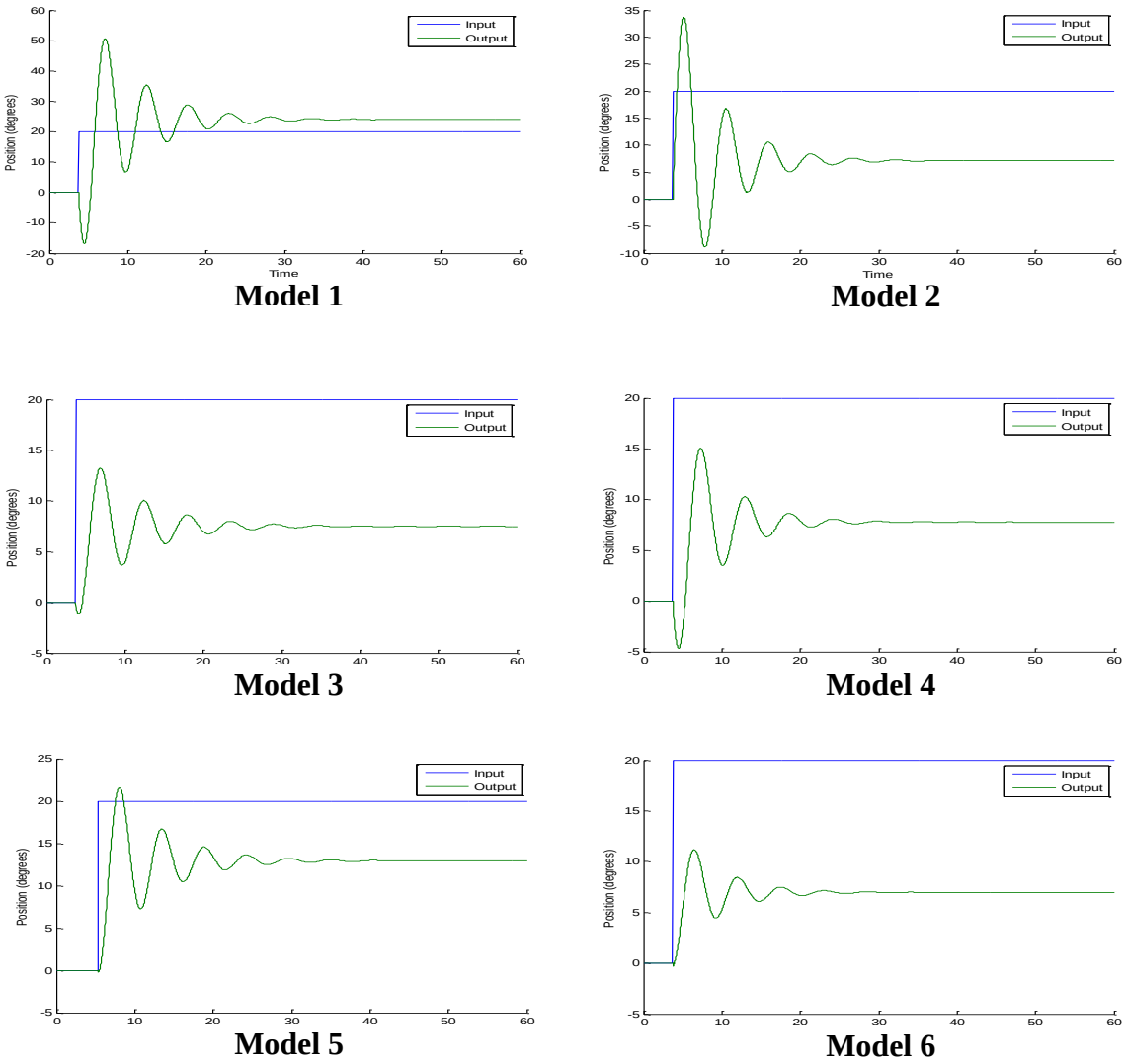


Figure 13: Simulations of all Six Models

Each model has an inherent DC Gain, or the ratio of the relationship between the magnitudes of the output compared to the input [6]. As noticed with Model 5, it did not reach the prescribed output of 20, but reached a steady-state value of approximately 14. To reach the desired output, a DC Gain must be applied to the output matrix C. In the case of Model 5, its exact DC Gain was determined to be 1.546. Figure 14 shows the difference between a model with and without DC gain implementation.

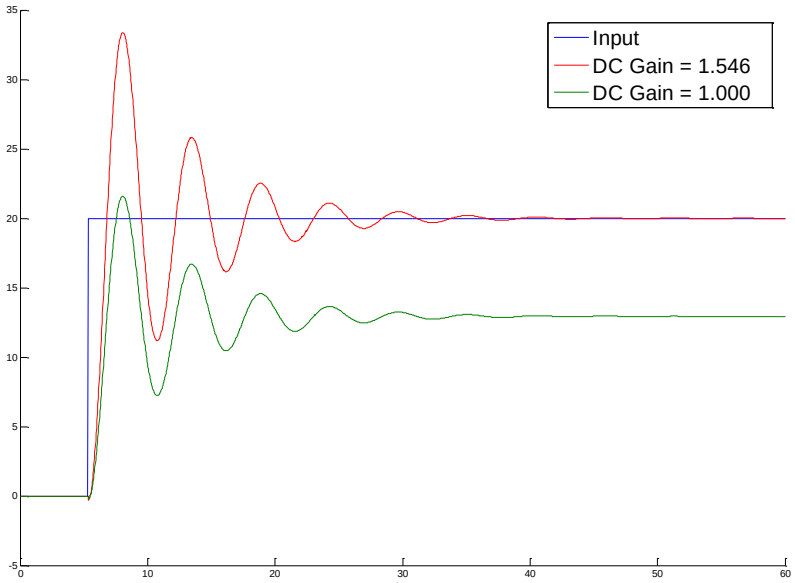


Figure 14: The Effects of DC Gain on the Buoy System

5.2 Animation

Besides the MATLAB code to simulate the state space models, code was also developed to provide visual feedback on the motion of the buoy. The code prepared a set of motion data using the experimentally derived model and prepared vertices for shapes to visually represent the payload and housing of the buoy. Next, the code ran an animation loop drawing each orientation of the buoy at each data point of the motion data, as shown by the still shots in Figure 15.

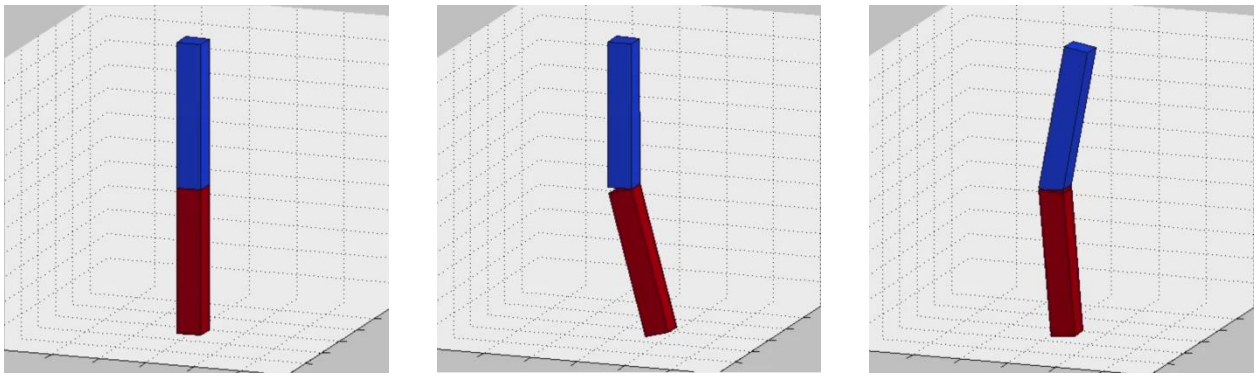


Figure 15: Buoy Animation at Time 0, 3, and 30 seconds

Chapter 6: Results

6.1 Validity of Experimental Model

The goal of this project was to derive and experimental model of the buoy to better understand the dynamics of the buoy, create an animation, prepare control efforts to keep the buoy vertical. The best way to determine the validity of the model was to test the model against the recorded experimental buoy orientation, as is shown in Figure 16. While any variation in actual buoy orientation versus the modeled orientation is within the design specification of five degrees, the greatest disparity is within the oscillation frequency. The actual buoy oscillated around 4.5 Hz, while the experimental model tended to oscillate near 5.3 Hz. Additionally, its took the experimental model nearly 30 seconds before it fell an entire cycle behind the real buoy. For the control purposes of this model, the first second or two after the step input are the most important. While the model is not perfect in those first couple of seconds, it performs remarkably well to match the amplitude and frequency of the first peak.

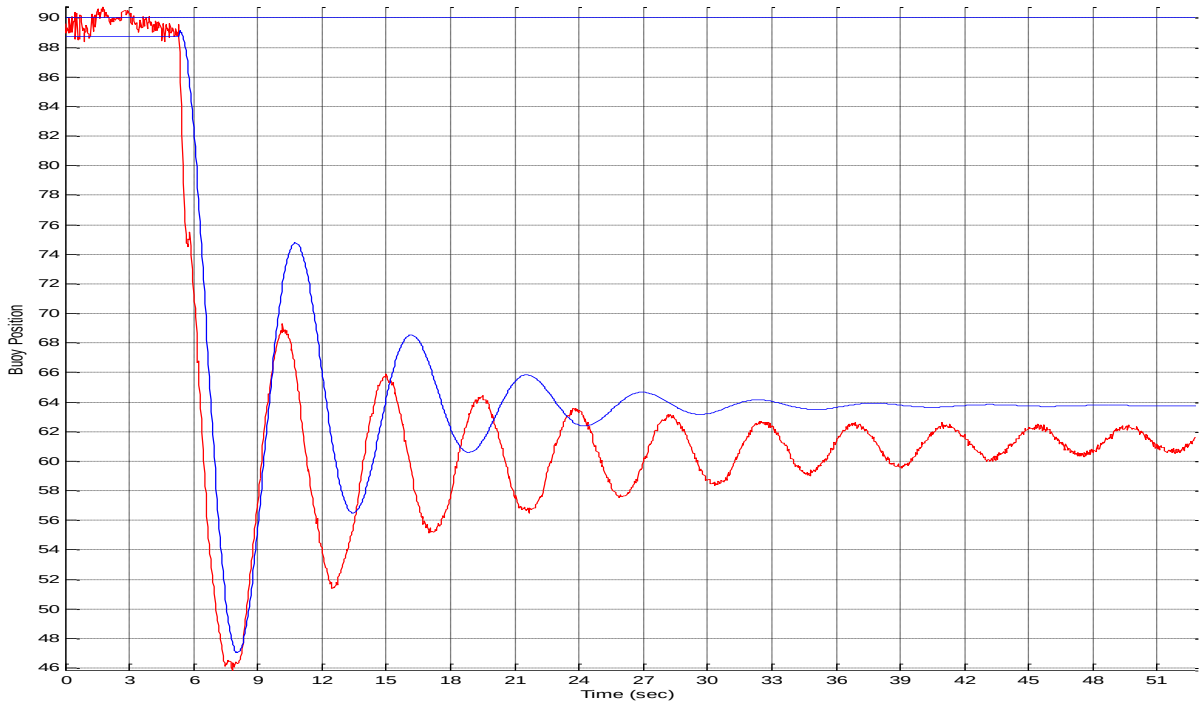


Figure 16: Model Prediction (Blue) vs. Buoy Orientation (Red)

6.2 Controllability and Control Law

Not all systems are capable of being controlled, but there is a process requiring matrix math to determine if a system is controllable [6]. Using the A and B matrices, concatenate the matrices following the method in Equation 3:

$$Cont = [B \quad AB \quad A^2B \quad A^3B \quad A^4B] \tag{3}$$

If the created matrix is of full rank, meaning all rows and columns are linearly independent, then the system it is possible to control all the states of the system. When the experimental model was tested, the controllability matrix was of full, and thus proving the experimental model was controllable.

A basic state variable feedback control model, as shown in Appendix G, was theorized for the buoy system [6]. This control method only required a control matrix K as a gain for the feedback loop and a control value of K_0. The constant K was calculated using Ackermann's formula for a fifth order state space model and the A, B, and C matrices from Model 5, and K_0 is the inverse of the DC Gain [6].

6.3 Theoretical vs. Experimental Model

As mentioned earlier, one of the easiest ways to compare state space models is to look at their eigenvalues. Figure 17 shows the eigenvalue plots for the theoretical model and the two servos from top to bottom overlaid on one another. Notice how each plot maintained a similar form with one pair of eigenvalues at the origin, one pair of eigenvalues left of the origin and farther from the x-axis and a real eigenvalue far left of the origin on the x-axis. While not all the original plots were the same scale, the overall similarity in each set of eigenvalue's silhouette depicted the similarity in the true nature of each model. Additionally, the theoretical model has eight eigenvalues and the experimental models only have five eigenvalues each to remain consistent with the order of each system as discussed earlier; this explains the two eigenvalues on the real axis and the extra pair of complex eigenvalues for the Theoretical model (black).

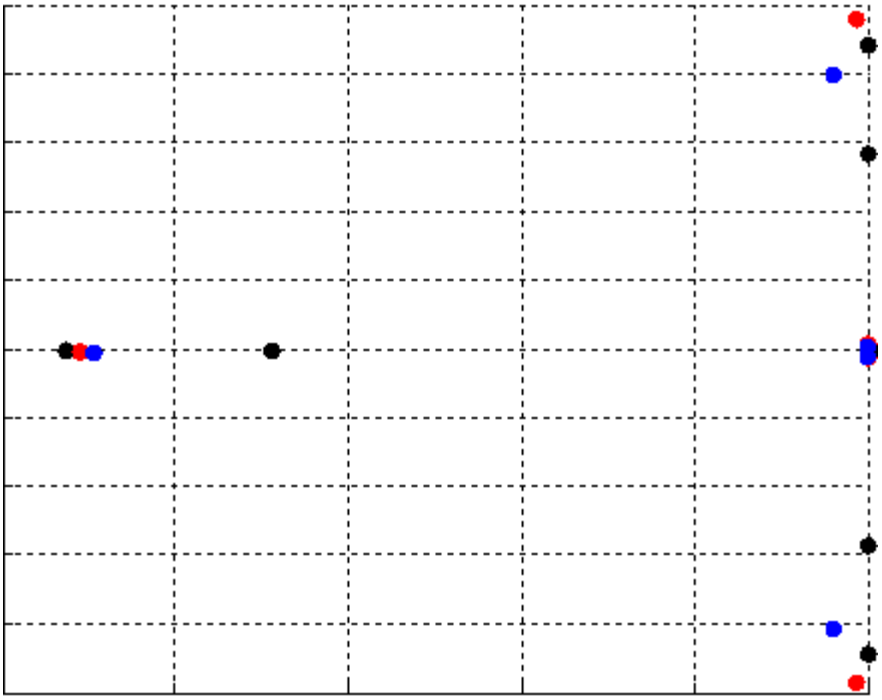


Figure 17: Comparison of Theoretical Model (black), Top Servo (blue), and Bottom Servo (red) Eigenvalues

Chapter 7: Future Work

7.1 Improvements to System Identification

While this project has been successful, there will always be room for improvement. Primarily, a functional on board sensor must be included in the upgraded version of the buoy to use for any future system identification or control efforts. Having a more precise measuring device like the IMU with a smaller sampling interval will help future experimentation and control efforts in two ways. First, it will reduce the unnecessary noise picked up using computer vision. It is difficult to get precise data with little noise when using computer vision because the reflected brightness changes sporadically at times, making the data less precise. Second, a quicker sampling interval will also help increase accuracy through more data points. The camera used for recording could only operate at 30 Hz (or frames per second). The IMU intended operated near 200 Hz, over six times quicker. With over six times the data points for the same testing will make the experimental models more accurate and precise to real world conditions.

7.2 Control Laws

The control law mentioned was purely theoretical because without an onboard orientation sensor, it was impractical to implement the control law. Implementing control efforts using computer vision would have required much more complexity than was necessary, introduced noticeable lag to the control system, and would not have met the design goal of creating a free-floating buoy. Implementing the control law on the buoy will be the next logical step after refining the model. There are many different control theories that could be tested on the buoy to determine the best combination of precision and agility to keep the buoy vertical at all times.

References

[1] L. K. Timothy. Method and Apparatus For An Improved Antenna Tracking System Mount On An Unstable Platform. *U.S. Patent No. 6433736B1,* August 2006.

[2] K. Doherty D. Frye and A. Hinton. Design and Evaluation of a Directional Antenna for Ocean Buoys. Technical Report WHOI-97-16, Woods Hole Oceanographic Institute, Woods Hole, MA.

[3] E. Briguglio. Antenna Stabilizing Buoy. *U.S. Statutory Invention Registration No. H1051*, May 1992.

[4] M. Romano and B. N. Agrawal. Attitude Dynamics/Control of Dual-Body Spacecraft With Variable-Speed Control Moment Gyros. *Journal of Guidance, Control, and Dynamics*, 27(4):513-525, 2004.

[5] R. S. Cortesi. *Modeling and Control of Joint-Actuated Buoys*. The George Washington University, 2012, 299 pages; 3502913.

[6] Nise, Norman S. *Control Systems Engineering*. 4th ed. Hoboken, NJ: Wiley, 2004.

Appendix A: Arduino Code to Read IMU Data

```
// Eric Fugleberg
// 1 August 2013

/*
Code to have Arduino Mega and VN-100 IMU
communicate to extract data to determine
yaw, pitch, and role of IMU. The code first sets
asynchronous data output to "no
output," then defines the reference frame rotation,
then (in accordance with the
user manual) saves the settings and restarts the IMU.
Once all this preliminary
work has been done, yaw, pitch, and roll data is
requested from the IMU is parsed
and sent the computer to view in the Serial Monitor.

Operating Settings for IMU:
1) Asynchronous Data Output OFF
2) Reference Frame set to (0,1,0,0,0,1,1,0,0)

*** IF HAVING PROBLEMS WITH ASYNC
DATA OR REFERENCE
FRAME CHECK FUNCTIONS AT BOTTOM OF
CODE ***

*/

//====================================
====================================

// ===== Declarations =====
// Include statements
#include <stdio.h>
#include <String.h>
#include <Servo.h>

// Serial3 to RECEIVE data from IMU into Arduino
#define IMUport Serial3

// Serial to SEND data from Arduino to PC (using
XBEE)
#define XBEEport Serial1

// Create Servo object
Servo botServo; // (Bottom Servo has the 3 grey
connectors near it)
Servo topServo; // (Top Servo has the Red and White
cables near it)

// ===== Declare more variables for data
manipulation =====
// IMU Command matrix and length
char IMUcmd[100];

// IMU Response String and length
String IMUres = "";
char IMUres_char;

// Declar yaw, pitch, and roll
String yaw;
String pitch;
String roll;

// Declare Quaternion yaw, pitch, and roll
float qyaw;
float qpitch;
float qroll;

// Declare quaternion strings
String Q0;
String Q1;
String Q2;
String Q3;

// Declare quaternion buffers
char q0[10];
char q1[10];
char q2[10];
char q3[10];

// Declare final quaternion matrix
float q[4];

// Radians to Degree constant
const float pi = 3.14159265359;
const float r2d = 180.0/pi;

// Servo center for buoy joint (BC: Bottom Center;
TC: Top Center)
// ***TEST TO CHECK VALUES ARE
CORRECT***
int BC = 85;
int TC = 92;
```

```
// Farthest joint can move off of Center (in degrees)
int most = 25;

// Timing element
float tot = 0;
float time = 0;

//====================================
====================================

// ===== Setup loop =====
void setup()
{
  // ===== Set Baud Rate for Serial3 and Serial1
=====
  openBaud();

  // ===== Set Async Data Output to Nothing =====
  setAsync();

  // ===== Set the Reference Frame Rotation (RFR)
=====
  setRef();

  // ===== Save the New Settings =====
  save();

//  // ===== Reset the IMU =====
//  reset();

  // ===== Start Data Request =====
  //startYPR();
  startQuaternion();
}

//====================================
====================================

// ===== Main program loop =====
void loop()
{
  // ===== Continual Data Request =====
  //getYPR();
  getQuaternion();

  // ===== Send Commands to the Servo Motors
=====
  //angles2servo();

  // ===== Clear the Strings =====
  clearStrings();
}

//====================================
====================================

// ===== FUNCTIONS =====


// Function:
// Open Serial Ports Serial3 and Serial1 to Baud
115200
void openBaud()
{
  // Open Serial Ports
  IMUport.begin(115200);
  XBEEport.begin(57600);

//  // Open pins for Servos
//  topServo.attach(44);
//  botServo.attach(46);
}


// Function:
// Set Asynchronous Data Output to Nothing
void setAsync()
{
  XBEEport.println("Setting Async Data...");
  strcpy(IMUcmd,"$VNWRG,06,0*6C");
  IMUport.println(IMUcmd);

  // Let Buffer Build
  delay(20);

  // Read buffer while there is data on it
  while (IMUport.available() > 0)
  {
    IMUres_char = IMUport.read();
    IMUres.concat(IMUres_char);

    if (IMUres_char == 'NUL')
      {
        break;
      }
  }

  // Remove leading or trailing white space
```

```
 IMUres.trim();

 // Check to see if response is correct
 if (IMUres == IMUcmd)
 {
  XBEEport.println("Async Data Turned Off\n");
  IMUres = "";
 }
 else
 {
  XBEEport.println("Async Data Not Turned Off,
Reset Arduino\n\n");
  while (1) {};
 }
}



// Function:
// Set Reference Frame
void setRef()
{
 XBEEport.println("Setting Reference Frame...");

strcpy(IMUcmd,"$VNWRG,26,0,1,0,0,0,1,1,0,0*6F"
);
 IMUport.println(IMUcmd);

 // Let Buffer Build
 delay(20);

 // Read buffer while there is data on it
 while (IMUport.available() > 0)
 {
  IMUres_char = IMUport.read();
  IMUres.concat(IMUres_char);

  if (IMUres_char == 'NUL')
    {
     break;
    }
 }

 // Remove leading or trailing white space
 IMUres.trim();

 // Check to see if response is correct and reset String
 if (IMUres == IMUcmd)
 {
  XBEEport.println("Reference Frame Properply
Set\n");
  IMUres = "";
 }
 else
 {
  XBEEport.println("Reference Frame Not Properly
Set, Reset Arduino\n\n");
  while (1) {};
 }
}



// Function:
// Save Settings
void save()
{
 XBEEport.println("Saving Settings...");
 strcpy(IMUcmd,"$VNWNV*57");
 IMUport.println(IMUcmd);

 // Let Buffer Build (500 necessary due to
mechanical specs)
 delay(500);

 // Read buffer while there is data on it
 while (IMUport.available() > 0)
 {
  IMUres_char = IMUport.read();
  IMUres.concat(IMUres_char);

  if (IMUres_char == 'NUL')
    {
     break;
    }
 }

 // Remove leading or trailing white space
 IMUres.trim();

 // Check to see if response is correct
 if (IMUres == IMUcmd)
 {
  XBEEport.println("Settings Saved\n");
  IMUres = "";
 }
 else
 {
  XBEEport.println("Setting Not Saved, Reset
Arduino\n\n");
```

```
   while (1) {};
  }
}


// Function:
// Reset IMU
void reset()
{
 XBEEport.println("Resetting IMU...");
 strcpy(IMUcmd,"$VNRST*4D");
 IMUport.println(IMUcmd);

 // Let Buffer Build
 delay(20);

 // Read buffer while there is data on it
 while (IMUport.available() > 0)
 {
  IMUres_char = IMUport.read();
  IMUres.concat(IMUres_char);

  if (IMUres_char == 'NUL')
    {
     break;
    }
 }

 // Remove leading or trailing white space
 IMUres.trim();

 // Check to see if response is correct
 if (IMUres == IMUcmd)
 {
  XBEEport.println("IMU Reset and Ready to
Roll!\n");
  IMUres = "";
 }
 else
 {
  XBEEport.println("IMU Not Reset, Reset
Arduino\n\n");
  while (1) {};
 }
}


// Function:
// Start Data Collection Process
void startYPR()
{
 // Starting to request yaw, pitch, and roll data
 XBEEport.println("Requesting Yaw, Pitch, and
Roll...");

 //Request yaw, pitch, and roll data and wait for
buffer to build
 strcpy(IMUcmd,"$VNRRG,8*4B");
 IMUport.println(IMUcmd);

 // Let buffer build
 delay(100);
}


// Function:
// Start Quaternoin collection process
void startQuaternion()
{
 // Starting to request yaw, pitch, and roll data
 XBEEport.println("Requesting Quaternions...");

 //Request yaw, pitch, and roll data and wait for
buffer to build
 strcpy(IMUcmd,"$VNRRG,9*4A");
 IMUport.println(IMUcmd);

//  // Center the buoy joint
//  topServo.write(TC);
//  botServo.write(BC);
//
//  delay(10);
}


// Function:
// Request YPR Data (yaw, pitch, roll)
void getYPR()
{

 // Wait for transmission of outgoing data to be
complete
 IMUport.flush();

 //Request yaw, pitch, and roll data and wait for
buffer to build
 strcpy(IMUcmd,"$VNRRG,8*4B");
 IMUport.println(IMUcmd);
```

```
  // Let Buffer Build
  delay(10);

  // Read buffer while there is data on it
  while (IMUport.available() > 0)
  {
   IMURes_char = IMUport.read();
   IMURes.concat(IMURes_char);

   if (IMURes_char == 'NUL')
     {
      break;
     }
  }

  // Remove leading or trailing white space
  IMURes.trim();

  // Parse Data
  yaw = IMURes.substring(10,18);
  pitch = IMURes.substring(19,27);
  roll = IMURes.substring(28,36);

  // Display data on computer
  XBEEport.print("Yaw: ");
  XBEEport.print(yaw);
  XBEEport.print("  Pitch: ");
  XBEEport.print(pitch);
  XBEEport.print("  Roll: ");
  XBEEport.println(roll);
}


// Function:
// Get Quaternion values and convert to pitch, roll,
and yaw
int getQuaternion()
{
  // Wait for transmission of outgoing data to be
complete
  IMUport.flush();

  // Start clock
  time = millis();

  //Request yaw, pitch, and roll data and wait for
buffer to build
  strcpy(IMUcmd,"$VNRRG,9*4A");
  IMUport.println(IMUcmd);

  // Let Buffer Build
  delay(10);

  // Read buffer while there is data on it
  while (IMUport.available() > 0)
  {
   IMURes_char = IMUport.read();
   IMURes.concat(IMURes_char);

   if (IMURes_char == 'NUL')
     {
      break;
     }
  }

  // Remove leading or trailing white space
  IMURes.trim();

//  // Print String to screen
//  XBEEport.print("Received: ");
//  XBEEport.println(IMURes);

  // Parse String into the 4 Quaternions
  Q0 = IMURes.substring(10,19);
  Q1 = IMURes.substring(20,29);
  Q2 = IMURes.substring(30,39);
  Q3 = IMURes.substring(40,49);

  // Character to Integer
  q[0] = convertStringtoFloat(Q0);
  q[1] = convertStringtoFloat(Q1);
  q[2] = convertStringtoFloat(Q2);
  q[3] = convertStringtoFloat(Q3);

  // Pitch, roll, yaw (degrees)
  qpitch = -r2d*atan(2*(q[1]*q[2] +
q[0]*q[3])/(q[3]*q[3] + q[2]*q[2] - q[1]*q[1] -
q[0]*q[0]));
  qroll = r2d*asin(-2*(q[0]*q[2] - q[1]*q[3]));
  qyaw = r2d*atan2(2*(q[0]*q[1] +
q[3]*q[2]),(q[3]*q[3] - q[2]*q[2] - q[1]*q[1] +
q[0]*q[0]));

  // Display data on computer
//  XBEEport.print("Pitch: ");
  XBEEport.print(qpitch);
  XBEEport.print("    ");
```

```
//  XBEEport.print(" Roll: ");
  XBEEport.print(-qroll);
  XBEEport.print("    ");
//  XBEEport.print(" Yaw: ");
//  XBEEport.println(qyaw);
  time = millis()-time;
  tot = tot + time;

  XBEEport.print(" ");
  XBEEport.println(tot);
}


// Function:
// Convert Strings to Floating points
float convertStringtoFloat(String convert)
{
  char test_as_char[convert.length()+1];
  convert.toCharArray(test_as_char,
convert.length()+1);
  float myFloat = atof(test_as_char);
  return myFloat;
}


// Function:
// Take Quaternion angles and send to Servos
void angles2servo()
{
  // Send commands to servo
  int tpos = TC-qpitch;
  int bpos = BC+qroll;

  // Make sure angles aren't greater than Max
  if (tpos >= (TC+most))
  {
    tpos = TC+most;
  }

  if (bpos >= (BC+most))
  {
}
  bpos = BC+most;
 }

 // Make sure angles aren't less than Min
 if (tpos <= (TC-most))
 {
   tpos = TC-most;
 }

 if (bpos <= (BC-most))
 {
   bpos = BC-most;
 }

 time = millis()-time;

 XBEEport.print(tpos);
 XBEEport.print(" ");
 XBEEport.print(bpos);
 XBEEport.print(" ");
 XBEEport.println(time);

 // Send commands to the Servo Motors
 topServo.write(tpos);
 botServo.write(bpos);
}


// Function:
// Clear all Strings
void clearStrings()
{
  IMUres = "";
  yaw = "";
  pitch = "";
  roll = "";
  Q0 = "";
  Q1 = "";
  Q2 = "";
  Q3 = "";
```

Appendix B: MATLAB Code to Perform Computer Vision Analysis of Video

```
% Eric Fugleberg
% Trident Project
% Take video and do frame by frame analysis
% Modified code from MATLAB Wiki
% ------------------------------------------------
% MODIFY LINES 18, 19, 71, 116, & 143 FOR DIFFERENT VIDEOS
% ------------------------------------------------

clc;
clear;
close all;
fontSize = 14;

% Set directory at proper level
cd('F:\Trident\Video\Top Servo');

%Buoy Movie file to use
buoyFile = 'step+15.avi';
videoTake = 'step+15';

% Open the Buoy video file
folder = fullfile('F:\Trident\Video\Top Servo');
movieFullFileName = fullfile(folder, buoyFile);

% Input video file into MATLAB
buoy = VideoReader(buoyFile);

% Determine how many frames and time of video
nFrames = buoy.NumberOfFrames;
time = buoy.Duration;
nFramesWritten = 0;

%% Make a .jpg for each frame of video
% Extract out the various parts of the filename
[folder, baseFileName, extentions] = fileparts(movieFullFileName);
% Make up a special new output subfolder for all the separate
% movie frames that we're going to extract and save to disk.
folder = pwd;   % Make it a subfolder of the folder where this m-file lives.
outputFolder = sprintf('%s/Frames from %s', folder, baseFileName);
% Create the folder if it doesn't exist already.
if ~exist(outputFolder, 'dir')
   mkdir(outputFolder);
end

% Loop through the movie, writing all frames out
% Each frame will be in a separate file with unique name
```

```
for frame = 1 : nFrames
    % Extract the frame from the movie structure.
    thisFrame = read(buoy, frame);

    % Convert from RGB to YCbCr
    thisFrame = rgb2ycbcr(thisFrame);

    % Write the image array to the output file, if requested
    % Construct an output image file name
    outputBaseFileName = sprintf('Frame %4.4d.jpg', frame);
    outputFullFileName = fullfile(outputFolder, outputBaseFileName);
    imwrite(thisFrame, outputFullFileName, 'jpg');

    % Update user with the progress.  Display in the command window.
    fprintf('Wrote frame %4d of %d.\n', frame, nFrames);
end

disp('~Video Transfered~');
close all;

%% Read each file and analyze
% Change directory to video pictures
cd('F:\Trident\Video\Top Servo\Frames from step+15');

% Do the analysis of the photos
p = which('Frame 0001.jpg');
filelist = dir([cd filesep 'Frame *.jpg']);
fileNames = {filelist.name};

for i = 1:nFrames
    im = imread(fileNames{i});

    % Orange Color Strip
    imbw=im(:,:,1)<137 & im(:,:,1)>88 & im(:,:,2)<102 & im(:,:,2)>78 & im(:,:,3)<180 & im(:,:,3)>145;

    % Green Color Strip
    % imbw=im(:,:,1)<80 & im(:,:,1)>45 & im(:,:,2)<135 & im(:,:,2)>115 & im(:,:,3)<135 & im(:,:,3)>110;

    % Filter out smaller shapes
    imbw2=bwareaopen(imbw,2400);
%     imshow(imbw2);
%     shg;

    % Get orienation of large objects
    stats = regionprops(imbw2,'Orientation');
    if (stats.Orientation < 0)
```

```
        ort(i) = stats.Orientation + 180;
    else
        ort(i) = stats.Orientation;
    end

    % Update user with the progress.  Display in the command window.
    fprintf('Analyzed frame %d of %d.\n', i, nFrames);
end

% Exit out of directory
cd('..');


% Close all figures and update user analysis complete
close all;
disp('~Frame Analysis Complete~');

% Update user of what's going on
disp('~Plotting Buoy Position~');
%% Use ort matrix and plot position of buoy
% First frame to use for modeling
frameone = 43;

% Make orientation matrix vertical
pos = ort(frameone:nFrames)';

% Calculate frames per second
fps = (nFrames)/time;

% Assign a time stamp for each frame
t = [(1/fps):(1/fps):time]';

% Plot the buoy position
figure(1);
hold on;
plot(t,ort','Color','red','LineWidth',0.5);

% Formatting to make plot look better
axis([0 time min(ort) max(ort)]);
refline(0,90);
xlabel('Time (sec)');
ylabel('Buoy Position');
set(gca,'YTick',[0:2:180]);
set(gca,'XTick',[0:3:time]);
set(gca,'YGrid','on');
set(gca,'XGrid','on');
```

Appendix C: MATLAB Code to Estimate State Space Model

```
% State Space Model Estimation from Data

% VDeclare Variables
step = 15;

% Declare necessary variables
yId = pos;
uId = step*ones((nFrames-frameone+1),1);
dt = (1/fps);
t = [(1/fps*frameone):(1/fps):time]';

% iddata Object
dataId = iddata(yId,uId,dt);

% Order of system
nx = 5;

%  State Space model (Continuous time domain
model)
[SSmodel,x0] = ssest(dataId,nx)

% Plot the Model over the actual data
A = SSmodel.a;
B = SSmodel.b;
C = SSmodel.c;
D = SSmodel.d;

sys = ss(A,B,C,D);
[y,t,x] = lsim(sys,uId,t,x0);

% Plot Simulation
plot(t,y,'--','LineWidth',1.5);

%% System Modal Transformation
% Create Modal form of Matrix
[modal, L] = canon(sys,'modal');

% Extract Modal Amod, Bmod, and Cmod
Amod = modal.a;
Bmod = modal.b;
Cmod = modal.c;

% Controlability Matrix
cont = ctrb(sys);

% Observability Matrix
obs = obsv(sys)
```

Top Servo Eigenvalues

-35	-30	-25	-20	-15	-10	-5
-0.238756025024234 - 1.217818550523391	-0.359865872197640 - 0.799455190408023i	-0.257316040819556 - 1.256734166826151	-0.227603285011642 - 1.079822316915961	-0.153139388442152 - 1.254108084371491	-0.127285208040843 - 1.272911308557431	-0.063707786101799 - 1.105029841821161
-0.238756025024234 + 1.217818550523391	-0.359865872197640 + 0.799455190408023i	-0.257316040819556 + 1.256734166826151	-0.227603285011642 + 1.079822316915961	-0.153139388442152 + 1.254108084371491	-0.127285208040843 + 1.272911308557431	-0.063707786101799 + 1.105029841821161
-15.53182332947720 - 68.40528984854741	-2.343192261071550 - 28.17669953317961	-0.351253907716265 - 47.26021592304421	-6.962451910014880 + 28.17652865030431	-9.869897944912690 + 106.9655371425871	-0.995869207553184 - 320.6688093601501	-0.556854948454777 - 51.71813820260131
-15.53182332947720 + 68.40528984854741	-2.343192261071550 + 28.17669953317961	-0.351253907716265 + 47.26021592304421	-6.962451910014880 - 28.17652865030431	-9.869897944912690 - 106.9655371425871	-0.995869207553184 + 320.6688093601501	-0.556854948454777 + 51.71813820260131
-7.110562217736500 + 0.000000000000000i	-82.57565871839830 + 0.000000000000000i	-11.41169111247010 + 0.000000000000000i	-17.19653228008260 + 0.000000000000000i	-20.35615434766930 + 0.000000000000000i	-1.984610928008240 + 0.000000000000000i	-19.56693147611490 + 0.000000000000000i

5	10	15	20	25	30	35
-0.168250270285065 - 1.153339830950361	-0.205547661714731 - 1.191782771370021	-0.206921467560955 - 1.228929261744221	-0.196565009687683 - 1.289087511890701	-0.160338155811341 - 1.287373382518501	-0.245561676640136 - 1.158859708445841	-0.335210086216574 - 1.118568483818401
-0.168250270285065 + 1.153339830950361	-0.205547661714731 + 1.191782771370021	-0.206921467560955 + 1.228929261744221	-0.196565009687683 + 1.289087511890701	-0.160338155811341 + 1.287373382518501	-0.245561676640136 + 1.158859708445841	-0.335210086216574 + 1.118568483818401
-0.897367816595974 - 64.85856915819641	-0.860030142459586 - 70.34174690039311	-4.067293888727901 - 12.18047479785671	-13.16031852841541 - 13.13549157622681	-54.02405071319611 - 58.87088474156481	-11.29412159757971 - 8.678819476629731	-2.624591773701261 - 38.06940379210331
-0.897367816595974 + 64.85856915819641	-0.860030142459586 + 70.34174690039311	-4.067293888727901 + 12.18047479785671	-13.16031852841541 + 13.13549157622681	-54.02405071319611 + 58.87088474156481	-11.29412159757971 + 8.678819476629731	-2.624591773701261 + 38.06940379210331
-27.82539355561900 + 0.000000000000000i	-14.49987415815310 + 0.000000000000000i	-289.5210401139470 + 0.000000000000000i	-196.9647797061070 + 0.000000000000000i	-8.996430915359950 + 0.000000000000000i	-521.0018716585610 + 0.000000000000000i	-9.213152449081330 + 0.000000000000000i

Bottom Servo Eigenvalues

-35	-30	-25	-20	-15	-10	-5
-0.238756025024234 - 1.217818550523391	-0.377337156017254 - 1.152884382166591	-0.175521828981087 - 1.269295341749511	-0.429310425678389 - 1.223947289477371	-0.413325349873731 - 1.156259884926721	-0.206240901117749 - 1.411123846738291	-0.173737564726641 - 1.489856881293381
-0.238756025024234 + 1.217818550523391	-0.377337156017254 + 1.152884382166591	-0.175521828981087 + 1.269295341749511	-0.429310425678389 + 1.223947289477371	-0.413325349873731 + 1.156259884926721	-0.206240901117749 + 1.411123846738291	-0.173737564726641 + 1.489856881293381
-15.53182332947720 - 68.40528984854741	-59.46122119748930 - 60.62346558234951	-20.71876525043360 - 67.69828790700981	-1.695101600434470 - 53.25534531968271	-5.234443389022030 + 25.93504286603631	-2.811375550157940 - 31.11144006533491	-1.165509805978250 - 74.93306116290931
-15.53182332947720 + 68.40528984854741	-59.46122119748930 + 60.62346558234951	-20.71876525043360 + 67.69828790700981	-1.695101600434470 + 53.25534531968271	-5.234443389022030 - 25.93504286603631	-2.811375550157940 + 31.11144006533491	-1.165509805978250 + 74.93306116290931
-7.110562217736500 + 0.000000000000000i	-937.2654424932620 + 0.000000000000000i	-642.5073749237930 + 0.000000000000000i	-6863.288406842930 + 0.000000000000000i	-2270.122486184160 + 0.000000000000000i	-421.8253479175160 + 0.000000000000000i	-41.39718239966970 + 0.000000000000000i

5	10	15	20	25	30	35
-0.242659315652382 - 1.177679184611611	-0.108141774743741 - 1.207088514456071	-0.132577090761291 - 1.299733219615061	-0.163485913181086 - 1.285678775497681	-0.284444443444238 - 1.168992282246401	-0.216163628576478 - 1.161718060169171	-0.281755161829918 - 1.256448320652341
-0.242659315652382 + 1.177679184611611	-0.108141774743741 + 1.207088514456071	-0.132577090761291 + 1.299733219615061	-0.163485913181086 + 1.285678775497681	-0.284444443444238 + 1.168992282246401	-0.216163628576478 + 1.161718060169171	-0.281755161829918 + 1.256448320652341
-4.766172052343080 - 69.42020649719971	-11.58166531945460 - 135.7979258661301	-28.62269941584060 - 63.29854474207741	-4.273619838961690 - 8.870472104163981	-1.260202233979660 - 81.24604951472311	-73.30063212978910 - 137.2453264543281	-3.060675361723970 - 59.02389796969531
-4.766172052343080 + 69.42020649719971	-11.58166531945460 + 135.7979258661301	-28.62269941584060 + 63.29854474207741	-4.273619838961690 + 8.870472104163981	-1.260202233979660 + 81.24604951472311	-73.30063212978910 + 137.2453264543281	-3.060675361723970 + 59.02389796969531
-7.510025404261200 + 0.000000000000000i	-64.13671645588020 + 0.000000000000000i	-479.7587591296090 + 0.000000000000000i	-5592.067209458970 + 0.000000000000000i	-12.24958277366820 + 0.000000000000000i	-80.44597849752720 + 0.000000000000000i	-33.22704647678990 + 0.000000000000000i

Appendix E: MATLAB Buoy Animation Code

Buoy Animation Script

```
% Simple Buoy Animation
% Inputs:  t, pitch, alpha

clear;

% Generate simulation data
step = 15;
SysIdDemo_Fugleberg
close all;

% Import simulation data
alpha = uId;
pitch = yId;

% Block specification
Lx = 0.05;
Ly = 0.05;
Lz = 0.30;

%% Motion data
%Body-b
rb = [0.0*ones(size(t')), -0.025*ones(size(t')),
0.0*ones(size(t'))];    % Position data
Ab = [pitch/(57.29577), 0.0*ones(size(t')),
0.0*ones(size(t'))]; % Orientation data (x-y-z Euler
angle)

%Body-a
ra = [0.0*ones(size(t')),  0.025*ones(size(t')), -
0.0*ones(size(t'))];    % Position data
Aa = [pitch/(57.29577)+pi-alpha/(57.29577),
0.0*ones(size(t')), 0.0*ones(size(t'))]; % Orientation
data (x-y-z Euler angle)

n_time = length(t);

%% Compute propagation of vertices and patches
for i_time=1:n_time
   Ra = Euler2R(Aa(i_time,:));
   Rb = Euler2R(Ab(i_time,:));
   VertexDataa(:,:,i_time) =
GeoVerMakeBlock(ra(i_time,:),Ra,[Lx,Ly,Lz]);
   VertexDatab(:,:,i_time) =
GeoVerMakeBlock(rb(i_time,:),Rb,[Lx,Ly,Lz]);
   [Xa,Ya,Za] =
GeoPatMakeBlock(VertexDataa(:,:,i_time));
   [Xb,Yb,Zb] =
GeoPatMakeBlock(VertexDatab(:,:,i_time));
   PatchData_Xa(:,:,i_time) = Xa;
   PatchData_Ya(:,:,i_time) = Ya;
   PatchData_Za(:,:,i_time) = Za;
   PatchData_Xb(:,:,i_time) = Xb;
   PatchData_Yb(:,:,i_time) = Yb;
   PatchData_Zb(:,:,i_time) = Zb;
end

%% Begin Recording Animation
% Set movie properties
writerObj = VideoWriter('BuoyAnim_ALL','MPEG-
4');
writerObj.FrameRate = 30;
writerObj.Quality = 100;

% Start Movie
open(writerObj);

%% Draw initial figure
figure;
ha =
patch(PatchData_Xa(:,:,1),PatchData_Ya(:,:,1),Patch
Data_Za(:,:,1),'r');
hb =
patch(PatchData_Xb(:,:,1),PatchData_Yb(:,:,1),Patch
Data_Zb(:,:,1),'b');
set(ha,'FaceLighting','phong','EdgeLighting','phong');
set(ha,'EraseMode','normal');
set(hb,'FaceLighting','phong','EdgeLighting','phong');
set(hb,'EraseMode','normal');

% Axes settings
set(gca,'FontSize',10);

% Make it look fancy
axis vis3d equal;
AZ = 110; EL = 14;
view([AZ,EL]);
camlight;
grid on;
```

```
% Zoom out a little
xlim([-0.3,0.3]);
ylim([-0.3,0.3]);
zlim([-0.3,0.3]);

% Remove Axis labels
set(gca,'XTickLabel',[]);
set(gca,'YTickLabel',[]);
set(gca,'ZTickLabel',[]);

%% Animation Loop
for i_time=1:n_time
   set(ha,'XData',PatchData_Xa(:,:,i_time));
   set(ha,'YData',PatchData_Ya(:,:,i_time));
   set(ha,'ZData',PatchData_Za(:,:,i_time));
   set(hb,'XData',PatchData_Xb(:,:,i_time));
   set(hb,'YData',PatchData_Yb(:,:,i_time));
   set(hb,'ZData',PatchData_Zb(:,:,i_time));

   frame = getframe;
   writeVideo(writerObj,frame);
end

% End Movie
close(writerObj);
```

SysIdDemo Script

```
% System ID Demo
% G. Piper Oct 2013

step = -20;

% Orginal System
A = [-482.8777958   0   0   0   0; ...
   0   -6.414535334    67.85914131 0   0; ...
   0   -67.85914131    -6.414535334    0   0; ...
   0   0   0   -0.154147244    1.162646254; ...
   0   0   0   -1.162646254    -0.154147244];

B = [-1.796636178; ...
   -0.419917733; ...
   0.205319849; ...
   -0.107425178; ...
   0.16212014];

C = [2.587525028 0.271517048 -0.061837923
      3.383308511 2.139850801];

D = 0;

Gss = ss(A,B,C,D);      % State space

% Time vector
dt = 1/30;      % Sample time
t = 0:dt:60;

% Input vector
u = step*ones(size(t));   % Unit step
u(find(t<3)) = 0.0; % Offset step function

Ns = length(t);      % Number of samples
Nu = 1;              % Number of input channels
TYPE = 'RGS';          % Random Gaussian Signal
% TYPE = 'RBS'          % Random Binary Sequence
% TYPE = 'PRBS'         % Pseudo Random Binary Signal
% TYPE = 'SINE'         % Sum-of-sinusoid Signal
% uId = idinput([Ns Nu],TYPE);
% uVal = idinput([Ns Nu],TYPE);
uId = u';
uVal = u';

% System response
yId = lsim(Gss,uId,t);
yVal = lsim(Gss,uVal,t);

% Store data
dataId=iddata(yId,uId,dt);       % Identification data
dataVal=iddata(yVal,uVal,dt);     % Validation data

figure(1)
hold on;
plot(t,uId,t,yId)
legend('Input', 'Output')
xlabel('Time')
ylabel('Position (degrees)')
```

Euler2R Function

```
function R = Euler2R(A)

% Euler angle -> Orientation matrix
a1 = -A(1);
a2 = -A(2);
a3 = -A(3);

R1 = [1, 0, 0;
      0, cos(a1), -sin(a1);
      0, sin(a1), cos(a1)];

R2 = [cos(a2), 0, sin(a2);
      0, 1, 0;
      -sin(a2), 0, cos(a2)];

R3 = [cos(a3), -sin(a3), 0;
      sin(a3), cos(a3), 0;
      0, 0, 1];

R = R1*R2*R3;
```

GeoVerMakeBlock Function

```
function VertexData = GeoVerMakeBlock(Location,Orientation,SideLength)

r = Location;
R = Orientation;

Lx = SideLength(1);
Ly = SideLength(2);
Lz = SideLength(3);

VertexData_0 = [Lx*ones(8,1), Ly*ones(8,1), Lz*ones(8,1)]...
   .*[0,0,0;
      1,0,0;
      0,1,0;
      0,0,1;
      1,1,0;
      0,1,1;
      1,0,1;
      1,1,1];

n_ver = 8;

for i_ver=1:n_ver
   VertexData(i_ver,:) = r + VertexData_0(i_ver,:)*R';
end
```

GeoPatMakeBlock Function

```
function [PatchData_X,PatchData_Y,PatchData_Z] = GeoPatMakeBlock(VertexData)

Index_Patch = ...
   [1,2,5,3;
   1,3,6,4;
   1,4,7,2;
   4,7,8,6;
   2,5,8,7;
   3,6,8,5];

n_pat = 6;

for i_pat=1:n_pat

   PatchData_X(:,i_pat) = VertexData(Index_Patch(i_pat,:),1);
   PatchData_Y(:,i_pat) = VertexData(Index_Patch(i_pat,:),2);
   PatchData_Z(:,i_pat) = VertexData(Index_Patch(i_pat,:),3);
end
```

Appendix F: MATLAB Linearization Code

```
% Equations of Motion for Articulated 2-Body Buoy
% (XZ Planar motion only)
% G. Piper - Oct 2013

clear

%Define State Variables
syms roll pitch yaw      % Orientation of Body_b
syms alpha beta          % Relative orientation between Body_a & Body_b (Joint angles)
syms w1_b w2_b w3_b      % Angular velocity of Body_b
syms w1_ab w2_ab         % Relative angular velocity between Body_a & Body_b

syms x y z               % Inertial position of joint
syms Vx Vy Vz            % Inertial velocity of joint

%Define Inputs
syms alpha_cmd beta_cmd  % Commanded joint angles

% Define Parameters
syms rcm_a rcm_b         % Body CM locations wrt to Body Frames
syms rj_a rj_b           % Joint location wrt to Body Frames
syms m_a m_b             % Body_a & Body_b mass
syms J11_a J22_a J33_a J11_b J22_b J33_b  % Moment of Inertias about Body CMs
syms radius_a radius_b length_a length_b  % Radius & length of cylindercal bodies
syms CDf12 CDf3          % Coefficients of drag
syms CDm12 CDm3          % Coefficients of drag moment
syms CD                  % Coefficient of drag for a flat plate (yaw radial fins)
syms nfins               % Number of yaw damping radial fins
syms lfins wfins         % Length and width of fins
syms zeta_j wn_j         % Joint actuator damping and natural frequency

syms z0                  % Nominal depth

% Constants
syms PI G rho
I = eye(3,3);

%-------------------------
% Velocity vector
% Velocity = [Vx; Vy; Vz;];  % Inertial velocity of joint
% (XZ Planar motion)
Velocity = [Vx; 0; Vz;];  % Inertial velocity of joint


% Angular velocity vectors
%-------------------------
% Angular velocity vectors
% Omega_b =  [w1_b;  w2_b;  w3_b];   % Body-b wrt Inertial Frame
% Omega_ab = [w1_ab; w2_ab; 0];      % Body-b wrt a Body-b
% (XZ Planar motion)
Omega_b =  [0;  w2_b;  0];      % Body-b wrt Inertial Frame
Omega_ab = [0;  w2_ab; 0];      % Body-b wrt a Body-b
```

```
% Vector cross product matrices
OmegaHat_b = [ 0           -Omega_b(3)    Omega_b(2);
            Omega_b(3)    0           -Omega_b(1);
           -Omega_b(2)    Omega_b(1)    0];

OmegaHat_ab = [ 0           -Omega_ab(3)    Omega_ab(2);
             Omega_ab(3)    0           -Omega_ab(1);
            -Omega_ab(2)    Omega_ab(1)    0];

%-------------------------
% Coordinate Transformations
B = [ cos(alpha)   0   -sin(alpha);
     0           1    0;
     sin(alpha)   0    cos(alpha)];
Bt = transpose(B);
Bdot = B*OmegaHat_ab;

B_b = [ cos(pitch)   0   -sin(pitch);
       0            1    0;
       sin(pitch)   0    cos(pitch)];
B_bt = transpose(B_b);
Bdot_b = B_b*OmegaHat_b;

B_a = B_b*Bt;
B_at = transpose(B_a);

Omega_a = B*(Omega_b-Omega_ab);

%-------------------------
% Joint location wrt to Body Frames
Rj_a = [0; 0; rj_a];
Rj_b = [0; 0; rj_b];

% Body CM locations wrt to Body Frames
Rcm_a = [0; 0; rcm_a];
Rcm_b = [0; 0; rcm_b];

% Body CM locations wrt to Joint Frame
D_a = Rcm_a - Rj_a;
D_b = Rcm_b - Rj_b;

% Total mass of buoy system
m_ab = m_a+m_b;

% System CM location wrt to Body Frames
Rcmsys_a = (m_a*Rcm_a + m_b*(Rj_a + B *D_b))/m_ab;
Rcmsys_b = (m_b*Rcm_b + m_a*(Rj_b + Bt*D_a))/m_ab;

% System CM location wrt to Body CMs
R_a = Rcm_a - Rcmsys_a;
R_b = Rcm_b - Rcmsys_b;
```

```
Rdot_a = -(m_b*Bdot*D_b)/m_ab;
Rdot_b = -(m_a*transpose(Bdot)*D_a)/m_ab;

%-------------------------
% Moment of Inertias about Body CMs
J_a = [ J11_a  0      0;
        0      J22_a  0;
        0      0      J33_a ];

J_b = [ J11_b  0      0;
        0      J22_b  0;
        0      0      J33_b ];

% Moment of Inertia about System CM
Jcmsys_a = J_a + m_a*(transpose(R_a)*R_a*I - R_a*transpose(R_a));
Jcmsys_b = J_b + m_b*(transpose(R_b)*R_b*I - R_b*transpose(R_b));

JcmsysDot_a = m_a*(2*transpose(R_a)*Rdot_a*I - Rdot_a*transpose(R_a) - R_a*transpose(Rdot_a));
JcmsysDot_b = m_b*(2*transpose(R_b)*Rdot_b*I - Rdot_b*transpose(R_b) - R_b*transpose(Rdot_b));

J = Jcmsys_b + Bt*Jcmsys_a*B;
Jinv = inv(J);

%============================
% External Forces and Moments
%============================

% Buoyant Forces and Moments

% Assume body-a is always completely submerged
vol_a = PI*radius_a*radius_a*length_a; % Volume boby a
volSub_a = vol_a;
fvolSub_a = 1;
Rcb_a = [ 0; 0; length_a/2]; % Center of buoyancy in a-frame

% Assume body-b is always partially submerged
vol_b = PI*radius_b*radius_b*length_b; % Volume boby b
d = -(z - rj_b)/cos(pitch);  % Water line wrt b-frame
volSub_b = PI*radius_b*radius_b*d; % Volume boby a submerged
fvolSub_b = volSub_b/vol_b;
Rcb_b = [radius_b*radius_b*d*tan(pitch)/4;
         0
         d/2+radius_b*radius_b*d*tan(pitch)/4];


Fbuoyant_a = B_a*[0; 0; rho*G*volSub_a];
Fbuoyant_b = B_b*[0; 0; rho*G*volSub_b];
Fbuoyant = Fbuoyant_a + Fbuoyant_b;

Mbuoyant_a = cross((Rcb_a-Rcm_a),Fbuoyant_a);
Mbuoyant_b = cross((Rcb_b-Rcm_b),Fbuoyant_b);
Mbuoyant = Mbuoyant_a + Mbuoyant_b;
```

```
%----------------------------
% Gravity Forces and Moments

Fgravity_a = B_at*[0; 0; -m_a*G];
Fgravity_b = B_bt*[0; 0; -m_b*G];
Fgravity = Fgravity_a + Fgravity_b;

Mgravity_a = cross((Rcmsys_a-Rcm_a),Fgravity_a);
Mgravity_b = cross((Rcmsys_b-Rcm_b),Fgravity_b);
Mgravity = Mgravity_a + Mgravity_b;

%----------------------------
% Drag Forces and Moments

Fdrag_a = [ -fvolSub_a*CDf12*length_a*radius_a * Velocity(1);
        -fvolSub_a*CDf12*length_a*radius_a * Velocity(2);
                -CDf3*radius_a*radius_a * Velocity(3) ];

Fdrag_b = [ -fvolSub_b*CDf12*length_b*radius_b * Velocity(1);
        -fvolSub_b*CDf12*length_b*radius_b * Velocity(2);
                -CDf3*radius_b*radius_b * Velocity(3) ];

Fdrag = Fdrag_a + Fdrag_b;

Mdrag_a = [ -sign(Omega_a(1))*fvolSub_a*CDm12*length_a*radius_a*Omega_a(1)^2;
        -sign(Omega_a(2))*fvolSub_a*CDm12*length_a*radius_a*Omega_a(2)^2;
         -sign(Omega_a(3))*fvolSub_a*CDm3*length_a*radius_a*Omega_a(3)^2 ];

Mdrag_b = [ -sign(Omega_b(1))*fvolSub_b*CDm12*length_b*radius_b*Omega_b(1)^2;
        -sign(Omega_b(2))*fvolSub_b*CDm12*length_b*radius_b*Omega_b(2)^2;
         -sign(Omega_b(3))*fvolSub_b*CDm3*length_b*radius_b*Omega_b(3)^2 ];

Myawpass = [ 0;
         0;
         -sign(Omega_a(3))*(1/8)*nfins*CD*rho*wfins*Omega_a(3)^2*((radius_a+lfins)^4-radius_a^4) ];

% Mdrag_a = [ -fvolSub_a*CDm12*length_a*radius_a*Omega_a(1)^2;
%           -fvolSub_a*CDm12*length_a*radius_a*Omega_a(2)^2;
%            -fvolSub_a*CDm3*length_a*radius_a*Omega_a(3)^2 ];
%
% Mdrag_b = [ -fvolSub_b*CDm12*length_b*radius_b*Omega_b(1)^2;
%           -fvolSub_b*CDm12*length_b*radius_b*Omega_b(2)^2;
%            -fvolSub_b*CDm3*length_b*radius_b*Omega_b(3)^2 ];
%
% Myawpass = [ 0;
%           0;
%           -(1/8)*nfins*CD*rho*wfins*Omega_a(3)^2*((radius_a+lfins)^4-radius_a^4) ];

MdragT = [ 0; 0; 0 ];  % Assume no significant translation so translation drag
                % coupling into rotation can be ignored

Mdrag = Mdrag_a + Mdrag_b + Myawpass + MdragT;
```

```
Fext = Fbuoyant + Fgravity + Fdrag;
Mext = Mbuoyant + Mgravity + Mdrag;

%============================
% Joint Actuator Dynamics
%============================
OmegaDot_ab = -2*zeta_j*wn_j*Omega_ab +wn_j*wn_j*([0; alpha_cmd; 0]-[0; alpha; 0]);

%============================
% Rotational Dynamics
%============================

P1 = -OmegaHat_b*Jcmsys_b*Omega_b;
P2 = -Bt*Jcmsys_a*B*(OmegaHat_ab*Omega_b-OmegaDot_ab);
P3 = -(OmegaHat_b-OmegaHat_ab)*Bt*Jcmsys_a*B*(Omega_b-Omega_ab);
P4 = -JcmsysDot_b*Omega_b;
P5 = -Bt*JcmsysDot_a*B*(Omega_b-Omega_ab);

OmegaDot_b = Jinv*(P1+P2+P3+P4+P5+Bt*Mext);

% Omega_a = B*(Omega_b-Omega_ab);
OmegaDot_a = Bdot*(Omega_b-Omega_ab)+B*(OmegaDot_b-OmegaDot_ab);

w1_a = Omega_a(1);
w2_a = Omega_a(2);
w3_a = Omega_a(3);



%============================
% Translational Dynamics
%============================
OmegaHat_a = [ 0      -w3_a    w2_a;
            w3_a       0   -w1_a;
           -w2_a    w1_a    0];

Bdot_a = B_a*OmegaHat_a;

OmegaDotHat_a = [ 0              -OmegaDot_a(3)   OmegaDot_a(2);
               OmegaDot_a(3)   0              -OmegaDot_a(1);
              -OmegaDot_a(2)   OmegaDot_a(1)   0];

OmegaDotHat_b = [ 0              -OmegaDot_b(3)   OmegaDot_b(2);
               OmegaDot_b(3)   0              -OmegaDot_b(1);
              -OmegaDot_b(2)   OmegaDot_b(1)   0];

Q1 = -m_a*(Bdot_a*OmegaHat_a + B_a*OmegaDotHat_a)*D_a;
Q2 = -m_b*(Bdot_b*OmegaHat_b + B_b*OmegaDotHat_b)*D_b;
VelocityDot = (Q1+Q2+Fext)/m_ab;

%============================================
% Linearize Model
%============================================
```

```
% State derivative vector
f = [ OmegaDot_b(2); OmegaDot_ab(2); VelocityDot(1); VelocityDot(3); Omega_b(2); Omega_ab(2); Velocity(1);
Velocity(3)];

% State vector
X = [ w2_b; w2_ab; Vx; Vz; pitch; alpha; x; z];
STATES = {'w2_b' 'w2_ab' 'Vx' 'Vz' 'pitch' 'alpha' 'x' 'z'};

% Input vector
U = [alpha_cmd];
INPUTS = {'alpha_cmd'};

% Output vector
y = [ w2_b; pitch ];
OUTPUTS = {'w2_b' 'pitch'};

% Jacobians
Asym = jacobian(f,X);
Bsym = jacobian(f,U);
Csym = jacobian(y,X);
Dsym = jacobian(y,U);

%====================================
% Operating Conditions (nominal states)
w2_b = 0;
w2_ab = 0;
Vx = 0;
Vz = 0;
pitch = 0;
alpha = 0;
x = 0;
% z = z0;

%====================================
% Nominal Inputs
alpha_cmd = 0;

%====================================
% Parameter Values (Refer to BouySimInit.)
PI = 3.1416;
G = 9.80665;     % Gravitational Acceleration (m/sec2)
rho = 1000.1;    % Water Density (kg/m^3)
rcm_a = -.4;
rcm_b = 0.3;
rj_a = 0;
rj_b = 0;
m_a = 1.5;
m_b = 1.0;
J11_a = 0.01;
J22_a = 0.01;
J33_a = 0.03;
J11_b = 0.03;
J22_b = 0.03;
J33_b = 0.01;
```

```
radius_a = 0.04;
radius_b = 0.04;
length_a = 0.2;
length_b = 0.4;

CDf12 = 1;
CDf3  = 1;
CDm12 = 3;
CDm3  = 0.1;
CD = 0;
nfins = 0;
lfins = 0;
wfins = 0;
zeta_j = 1.01;
wn_j =  20.0;

%====================================
% Solve for equilibrium depth
fo = subs(f);
zo=solve(fo(4)==0);
z = double(zo)
%====================================

% Substitute values into A & B matrices
Asym = subs(Asym);
Bsym = subs(Bsym);
Csym = subs(Csym);
Dsym = subs(Dsym);

% Convert symbols into numbers
Asys = double(Asym);
Bsys = double(Bsym);
Csys = double(Csym);
Dsys = double(Dsym);


disp('//////////////////////')
disp('Linearized Buoy Model')
disp('//////////////////////')
sys = ss(Asys, Bsys, Csys, Dsys, ...
         'statename',STATES,...
         'inputname',INPUTS,...
         'outputname',OUTPUTS)
damp(Asys)

% Create Modal form of Matrix
[modal, L] = canon(sys,'modal');

% Extract Modal Amod, Bmod, and Cmod
Amod = modal.a;
Bmod = modal.b;
Cmod = modal.c;
```

Appendix G: State Variable Feedback

MATLAB Code

```
% State Plus Integral Control

clear;

% Declare all Matrices for system
A = [-482.8777958   0   0   0   0; ...
   0   -6.414535334    67.85914131 0   0; ...
   0   -67.85914131    -6.414535334    0   0; ...
   0   0   0   -0.154147244    1.162646254; ...
   0   0   0   -1.162646254    -0.154147244];

B = [-1.796636178; ...
   -0.419917733; ...
   0.205319849; ...
   -0.107425178; ...
   0.16212014];

C = 1.546*[2.587525028 0.271517048 -0.061837923 3.383308511 2.139850801];

% Desireable Poles based on design parameters
P = [-10+10i -10-10i -100+100i -100-100i -200];

% Calculate Control 'K' Value
K = acker(A,B,P);

% Run SIMULINK diagram
sim('Fugleberg_PID');

% Plot control plots
figure(1)
hold on;
plot(tout,pos);
xlabel('Time (sec)');
ylabel('Buoy Position (deg)');
```

SIMULINK Diagram